1991

A Framework for Intelligent Parallel Compilers

Ko-Yang Want

Report Number:
91-030

INTELLIGENT PROGRAM OPTIMAZATION AND PARALLELIZATION FOR PARALLEL COMPUTERS

Ko-Yang Wang

CSD-TR-91-030
April 1991

A FRAMEWORK FOR INTELLIGENT PARALLEL COMPILERS

Ko-Yang Wang

Computing About Physical Objects,
Department Of Computer Sciences, Purdue University, West Lafayette, IN 47907
E-mail: kyw@cs.purdue.edu
Phone number: (317) 494-4465
Fax number: (317) 494-0739

ABSTRACT

In this paper, we outline obstacles in building high performance, intelligent parallel compilers and present some methodologies for overcoming these problems. A framework for contructing parallel compilers that can analyze and optimize programs automatically and intelligently is proposed. This framework utilizes sophisticated AI techniques to optimize programs in a systematic approach in order to prone non-promising decision subtrees as early as possible. Methodologies to apply this framework to build optimizing parallel compilers, including heuristic-guided state-space search and planning, machine knowledge manipulation, system-knowledge organization and inference, opportunistic reasoning, and a problem-solving model called hier-blackboard, are also discussed.

Keywords: *AI, architecture, compiler, heuristics, intelligence, parallelism, planning, problem-solving, program-transformation, optimization*

September 29, 1990

---

† Technical Report, CSD-TR-1044, CER-90-52, Department of Computer Sciences, Purdue University, November 1990.

ACKNOWLEDGMENTS

I would like to thank all the individuals who have contributed, both personally and professionally, to the completion of this thesis. In particular, I gratefully acknowledge the friendship, support and guidance of my major professors Piyush Mehrotra and Dennis Gannon. I furthermore would like to thank my committee members Professors Tim Korb and Mike Attallah for their valuable input and suggestions.

Much of the research reported in this thesis was done during 1985-1988 while I was a graduate student in the Computer Science Department at Purdue. However, the writing of this thesis was on and off during the last three years - including the last two and half years that I worked as a full-time research associate in the Parallel and Distributed Computing Group of the Computing About Physical Objects (CAPO) project. I would like to thank Dr. Houstis and Dr. Rice for their guidance and support; without their help and encouragement, I would never have finished the writing of this thesis. I thank all my colleagues in the CAPO for the enjoyable experience of working with a group of fine researchers. I would like to thank Dr. Greg Pfister and the RP3 group of IBM for giving me the opportunity of doing on-site research at IBM's facilities in Yorktown heights, New York.

I thank all my friends at the computer sciences department of Purdue University who make my stay a rewarding experience. I like to thank Dr. William Gorman for his support while I was a graduate student in the computer science department. I also like to thank the following nice secretaries for their help: Patty Minniear, Margaret Fahl, Daloris Williamson, Wilma Birge, Paula Perkins, Georgia Conarroe, and Candace Walters. Dr. Frank Oreovicz proofread the whole thesis and gave me lots of advice in technical writing which I am duly grateful. I also would like to thank Scott McFaddin who proofread chapters 3 and 4 and also gave many valuable suggestions.

I like to thank my wife Yuh-Jiun for her love, support and patience. I thank my daughter Alice and son Arthur who made my years at Purdue busy but meaningful. I also thank my mother for everything she rendered me. I thank my late-father and regret that he can't live to see the completion of this thesis.

TABLE OF CONTENTS

Page

Page

Page

CHAPTER 1
INTRODUCTION

The main focus of this thesis is the methodology for constructing intelligent, optimizing, parallel compilers and programming environments. We emphasize intelligence in the context of the optimization of program parallelism because of the following two reasons:

1. Program-parallelism optimization is a difficult task and high level intelligence is required.
2. Applying state-of-the-art AI techniques to parallel compiling may significantly increase the effectiveness and efficiency of parallel compilers. However, this approach has not received its due attention in the parallel compiler research community.

Two typical remarks that we received when we first proposed utilizing AI techniques to build intelligent parallel compilers back in 1985 were: "Aren't you making a difficult problem even more difficult?" and "How can you build an expert system for a field that has no experts?" Indeed, building parallel compilers or programming environments has never been straightforward. The task is so complex that no satisfiable parallel compilers exist today. However, the purpose of introducing AI techniques into parallel compiling is to derive new methodologies for organizing and integrating program optimization knowledge and a framework for systematic and automatic analysis so that intelligent behaviors can be observed in parallel compilers. As will be demonstrated in this thesis, powerful program optimization tools can be built based on some relatively simple methodologies. The second remark demonstrates a common misunderstanding about expert systems. In fact, the primary areas where expert system technology can be applied are ill-conditioned, complex problems where the general level of understanding about the field is not mature enough to solve the problem analytically. The problem of program parallelism optimization falls into this category.

In this chapter, we identify some problems with current state-of-the-art optimizing parallel compilers and programming environments and discuss new directions for solving these problems.

1.1. Motivation

An emerging trend in computer design is the reliance on parallelism to achieve performance improvements over sequential processors. During the last decade, parallel architectures have migrated out of the research laboratories and spread into various classes of commercial product lines that range from supercomputer class machines, such as the Cray-YMP and Intel Touchstone, to workstation class machines such as Transputers or NCUBE-4. The advantage of utilizing parallelism is obvious - concurrent execution provides the critical speed up that many applications need and opens up opportunities for many new applications that were not computable before. For supercomputers, parallelism provides a critical technology to break the sequential speed barrier. For general purpose machines, parallelism allows manufacturers to produce powerful computers with impressive peak performance and excellent price-performance ratio without exclusively relying on expensive, cutting edge hardware technology. This lowers the cost of the machines significantly.

However, parallelism is not without its price. Parallel programs contain multiple threads of the control and data flow. Many new concepts and difficulties that do not exist in programming sequential computers are encountered in programming parallel computers. Issues such as the correct order of concurrent data update and accesses, critical regions, communication, and deadlock prevention need to be handled carefully to ensure the correctness of the programs. Furthermore, complicated issues such as program partitioning, mapping, scheduling, cache and local memory utilization, and synchronization must be addressed to utilize the parallel capability of the machines. This makes programming parallel computers a very tricky and complex task. Programming parallel computers can be several orders of magnitude harder than programming their sequential counterparts [CCHKT88]. The efficiency of the program is usually tied closely to the structure of the target machine. Highly optimized programs are usually non-portable. Worse yet, it is normally very difficult to predict the

correct behavior or to assess the performance of the program on a target machine. Debugging parallel programs often becomes a nightmare for programmers.

Because of these difficulties, the mass parallelism promised by parallel architectures is rarely realized. Studies indicate that the effective performance of parallel computers usually ranges between only 5% to 35% of their peak performance [Bem86, Dong87]. Such poor delivered performance emphasizes the great need for appropriate software tools for parallelism utilization. What are needed are user-friendly parallel compilers or programming environments that can shield the difficulties of parallel programming from the users.

1.1.1. Parallel Programming Environments

A parallel programming environment is a software system designed to help the user to program and debug programs for parallel computers. It is usually a software system that is the integration of various subsystems such as programming tools (editors, visual programming tools, etc.), parser, program dependence analyzer, parallelism optimizer, code generators, performance evaluation tools, visualization tools, debugging tools, and, most importantly, a friendly user-interface. Different programming environments emphasize different aspects of the parallel programming task. We are particularly interested in programming environments that are capable of helping users optimize programs for efficient execution on different classes of parallel computers.

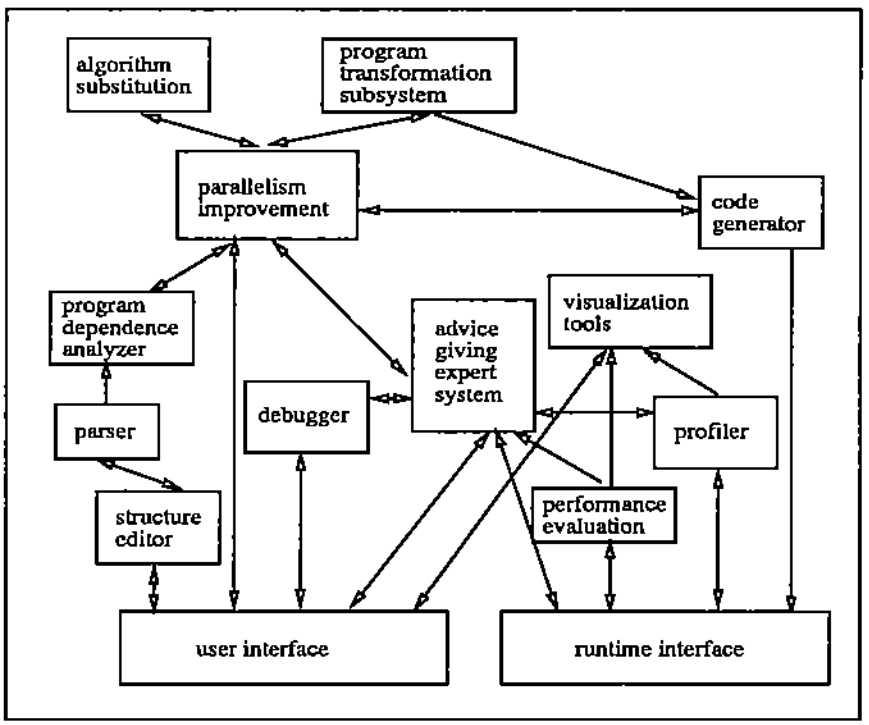

Figure 1.1. *The structure of a typical parallel programming environment.*

What an ideal parallel programming environment consists of is debatable. Ideally, the user should implement the application in a high level programming language without worrying about the efficiency problem and leave the optimization task to the compilers. In other words, the responsibility for mapping programs efficiently to the target architectures lies primarily with the compilers and programming environments instead of with the users. In this way, the programming environment can shield programmers from the pragmatic details of parallel programming and allow them to concentrate on the structure of the algorithms and the correctness of the programs. The use of this kind of programming environment can be best described by the following scenario.

> *After developing the algorithms for his problem, the programmer encodes his/her application in a machine-independent form (which can be either sequential or parallel). He then selects a set of target machines. For each of the target machines, the programming*

environment parallelizes the code and generates an optimized program that matches the underlying architecture model. Depending on his/her parallel programming experience, the programmer may select to interact with the system's decision-making process. Such interaction can range from querying the system for its reasons in making some particular decisions to providing suggestions or other input which would ease the task of the system. In the extreme case, the programmer may actually direct the choices taken by the system.

Of course the programming environment as described above does not exist today but there are some ongoing efforts in this direction. A good parallel programming environment should provide programming and optimization advice to ordinary parallel programmers so that user involvement in the decision-making is minimum.

1.1.2. The State-of-the-art of the Current Technology

Most parallel compilers and programming environments of today have the following problems:

- *They lack the needed intelligence to make critical decisions.* Most systems rely on the user to make critical optimization decisions.
- *They are inefficient.* Most systems are slow and expensive to use.
- *They are hard to improve.* Most systems lack a systematic organization of the system knowledge and often have ad hoc heuristics scattered throughout the system. This makes it difficult for the systems to evolve. None of the existing parallel compilers or parallel programming environments have knowledge acquisition facilities or learning capabilities.
- *They have a limited scope of applications.* Most existing systems can only be applied to a very limited set of target architectures. Transferring the knowledge in a parallel compiler for a particular machine to another requires significant effort.

As a result of these problems, most parallel compilers rely on the users to exploit opportunities for concurrent operations. Users have to structure their applications carefully in specific forms so that the compiler can easily recognize the parallelism available in the program. This means that users have to figure out most or all the transformations that the compilers need to perform and annotate the programs to instruct the compilers to generate efficient code. This is certainly not the kind of help that novice programmers expect from parallelizing compilers.

Although program vectorization is considered to be a well-researched and well-understood subject ([AlKe84a, AlKe87, KKLW80, Wolfe82]), research in program parallelization and optimization for non-shared memory or hierarchical multiprocessor architectures is still in its infancy. Only recently have extensive efforts been underway in this direction: Parafrase II at CSRD [PGHLLS89], PFC [AlKe84b] and Parascope [CCHKT88] at Rice, PTRAN at IBM [Allen86, ABCCF88], and our efforts in building intelligent parallel compilers [Wang85, WaGa89, Wang90d]. It is unfortunate that the advances in parallel hardware have not been met with needed advances in software technology. In fact, the slowness in market growth of parallel computers may be linked indirectly to the slowness in the development of good parallel compilers and programming environments. In the next section, we will discuss some difficulties in building such environments with current state-of-the-art technology and propose some solutions to the problems.

1.1.3. Obstacles in Building Parallel Programming Environments

Why is the parallel programming software still in such a primitive state even after extensive research efforts during the last decade? The major difficulties in generating powerful optimizing parallel programming environments can be categorized as follows:

1. *Lack of comprehensive understanding of parallelism utilization:* Despite extensive research during the past decade, we still do not have an overall picture of how parallelism can be optimized without extensive user involvement.
2. *Detailed knowledge about the target machines required for program optimization:* Different parallel architectures use different techniques to speed up computations and require different tricks to utilize the features. Extensive knowledge about the underlying hardware is needed.
3. *Dynamism in program behavior:* The performance of some programs depends on input and the control flow of the program.

4. *Incomplete knowledge at compile time:* Parallel compilers have to base their decisions on approximate information at compile time. These approximations are often incomplete, rough and rely on an unrealistic and simplified model of computation.
5. *Huge decision-trees for parallelism optimization:* Even for a program of medium size, the decision-tree for program optimization can be quite large. Methodologies for pruning unneeded branches in the decision-tree are needed.
6. *Use of ad hoc heuristics:* The techniques adopted by parallel compilers and programmers of parallel computers are mostly ad hoc heuristics. A number of heuristics are needed in order for the compiler to perform an adequate job. Unfortunately, most systems lack proper knowledge organization facilities to utilize the heuristics effectively and systematically. Also, these ad hoc heuristics are mostly non-portable.
7. *Expensive dependence analysis and performance estimation:* Program dependence information and estimation of performance are usually very computation-intensive.

Because of these difficulties, much of the burden of achieving high performance is shifted to the programmer. Most people agree that current parallel programming tools need a much higher degree of intelligence. The question is: How far away are we from building truly intelligent parallel compilers that can shield the user from optimization details? Have we effectively utilized the state-of-the-art technology in building the current generation of parallel compilers? Judging from the abundance of research results and the state of current parallel compilers, the answer is probably a "no." From our point of view, the root of the problem lies in the lack of systematic mechanisms for the reasoning and control of program parallelization and optimization process. Methodologies that can effectively integrate and utilize the current technology to improve parallel programming environments and compilers can have an immediate impact on the development of parallel software and should be important topics in the current research on parallel processing. In particular, combining advance program transformation techniques and the state-of-the-art AI technology in the construction of parallel compilers or programming environments is a very promising approach. Four important areas that need much more concentrated and coordinated research efforts are:

1. *Framework for systematic program analysis and intelligent program restructuring control.* This is the central topic of chapter 4.
2. *The integration of machine properties into the program restructuring process and the representation and manipulation of machine features.* These problems were discussed in [WaGa89, Wang88] and will be briefly discussed here. [Poly86] and [Sarkar87] discussed task creation and scheduling problems based on some machine features.
3. *Methodologies for analyzing, representing, organizing, and integrating heuristics for improving parallelism.* A framework for knowledge organization and control, called the *heuristic hierarchy*, is discussed in section 4.4.3, and the concept is extended to a new problem-solving model called the *hier-blackboard* (see section 4.4.4).
4. *Learning models and knowledge acquisition tools for the enhancement of the system knowledge.*

1.2. Problem Statements

This research is aimed at studying methodologies for constructing intelligent parallel compilers for different classes of parallel computers. We will examine new techniques and methodologies for building intelligent compilers for different classes of parallel architectures. Two key issues that we concentrate on are the intelligence of the system and the portability across a wide spectrum of target architectures.

More specifically, we intend to study the following problems:

1. *Paradigms for program restructuring.* Program restructuring techniques are only mechanical procedures to change the program structure; to effectively utilize these techniques, we need to couple the techniques with knowledge of the target machine and heuristics. A framework for systematically selecting program transformations and evaluating the results is needed. The efficiency, flexibility, and effectiveness of the framework need to be studied.
2. *Methodologies for improving the parallelism of the program.* Study the heuristics for applying program transformations to match the parallelism of the program with the target machine, the theoretic foundations of some program transformation techniques, and the incorporation of global consideration in the

decision-making process.

3. *Knowledge manipulation for parallel program optimization.* These include techniques for the analysis, acquisition, organization, integration and accumulation of the knowledge for program parallelization and optimization.

4. *Application of AI techniques to parallel compiling.* Assess possible contributions of various AI techniques in the construction of intelligent parallel compilers.

In our view, these problems are essential for solving the general problem of building parallel programming environments that we discussed above. To demonstrate the techniques developed in this thesis, a prototype system has been constructed. Some experimental results and comparisons with other parallel programming environments and parallel compilers will be presented.

1.3. Our Approach

Our goal is not to build a production quality parallel compiler that generates high quality object code for certain parallel computers; this would require a great implementation effort and overshadow the main ideas of this research. What we are trying to do is to derive methodologies and theoretical foundation for construction of such systems and show that these approaches are feasible. Our approach is based on a new framework for systematic analysis and restructuring of the program. This framework integrates machine features and performance evaluation into the decision-making process of program optimization. Under this framework, the program optimization process is an iterative process of finding a suitable program transformation sequence to improve the match between the program and the target machine. AI techniques can be utilized in this iterative process to prune non-promising branches of the decision tree, provide expert advice and explanation to novice users, and support learning to improve intelligence of the compiler.

This framework is realized by a generalized *blackboard* problem-solving model called the *hier–blackboard* which features opportunistic reasoning and hierarchical knowledge organization. Features of the target machine that affect the performance of the program are explicitly encoded in the heuristics. An object-oriented machine knowledge representation scheme is designed to support reasoning on the knowledge and integration of heuristics. A machine knowledge manipulation system is implemented to provide supports for the program optimization system and the knowledge acquisition tool.

A systematic program optimization process requires a flexible, efficient, and accurate performance prediction model. We have designed and implemented a performance prediction model to estimate effects of program transformations on the program. This performance prediction is based on the machine knowledge manipulation system and integrated with the systematic program restructuring process.

We also study heuristics for utilizing various transformation techniques and the theoretic foundation of two useful transformations, array reshaping and message consolidation.

The essential requirement for a system to be intelligent is the ability to acquire new knowledge or self-learning. A compiler that learns from its past experience will greatly enhance its capabilities and evolve with advances in the technology. Although no self-learning modules are implemented at this point, the whole system framework is designed with self-learning in mind. Provisions for integrating learning modules are supported in modules for knowledge organization, program optimization, and performance prediction.

To summarize, our approach has the following distinct features:

- The program optimization process is driven by the features of the target machine and the program.
- Machine knowledge manipulation is integrated with the decision-making, performance-prediction, and knowledge-manipulation processes.
- It supports multiple target machines and allows knowledge generalization and knowledge transfer.
- The whole decision tree is examined; AI techniques are utilized to prune non-promising branches of the decision-tree.
- Supports systematic manipulation (encoding, organization, integration, and utilization) of domain knowledge.
- New knowledge acquisition and self-learning modules can be easily incorporated.

The implementation of our approach differs from other parallel compilers in the following four ways. First, we are using second generation expert system techniques as part of the internal design. Second, we are

exclusively relying on heuristics to make program optimization decisions based on the features of the machine and the program. No prior sequence of transformation is required. Third, our system can handle different classes of target architectures. Adding new target machines is a matter of specifying features of the machines and installing new heuristics for programming the machine. Fourth, the level of user interaction can be chosen by the user. This allows the system to adjust itself to suit the different backgrounds of the users. Also, depending on the available resources, the level of optimization degree is also selectable. Unlike other optimization systems where the systems skip some optimization process altogether when the optimization degree is low, the optimization degree is actually an indication of how many resources should be devoted to do certain costly estimations of the performance. Therefore, the optimization degree will not affect the functionality of the optimization but may use cheap heuristics to obtain a rough estimation of the needed information.

Why do we choose to use the expert system approach and to support multiple target architectures? We examine this question in the following two subsections.

1.3.1. Expert System Technologies

Expert systems are suitable for problem domains with great complexity and jobs for which it is difficult to develop algorithms that can handle general cases. These kinds of problems are usually solved by employing human heuristics. Computer-based expert systems seek to capture enough knowledge of human specialists so that they can solve the problem. Due to the inherent difficulties of knowledge acquisition and representation, the discovery and test cycle of integrating the knowledge needs to be repeated many times before the system is powerful enough to solve general cases of the problem. With the gradual maturation of the knowledge about parallelism, the knowledge of the parallel compilers needs to be updated frequently. The open-ended characteristic of this process makes the expert system a natural choice for implementation.

First generation expert systems rely on the use of surface knowledge, such as simple heuristics. As a result, these expert systems show only a limited degree of intelligence. This is because human experts not only have a ground in the declarative and the procedural knowledge of their particular domains, but also possess something beyond facts and procedures. A major aspect of expertise resides in its structure and organization (deep knowledge). However, the know-how sought by expert systems does not come pre-packaged as an explicit and objective model. It may be better characterized as a collection of semi-complete, semi-structured, hedged, and subjective know-how [Hayes83]. Therefore, it is important that the knowledge acquisition techniques catch the underlying structure and organization of the knowledge. Second generation expert systems try to solve this problem by looking beyond the use of surface knowledge and utilizing two additional features: tools for acquisition of deep knowledge and machine learning. The expert system shell should integrate a variety of tools that allow for knowledge acquisition, explanation and utilization of the complex domain knowledge. The ExpShell discussed in chapter 8 represents our efforts in this direction. The importance of machine learning is in its low degree of human involvement. Although self learning is still not a reality now, it has much higher potential than human-assisted knowledge acquisition to learn in the long run. It can utilize the available computing power to explore more possibilities and is not limited by the knowledge of the human experts.

One major drawback of the rule-based expert systems is the opacity of the knowledge. Translating heuristics into rules causes the knowledge to be fragmented. Even though there are still strong relations between many of the rules, the fragmentation causes an unfortunate loss of coherence. We address this problem by deriving a new problem-solving model called the *hier-blackboard*. The hier-blackboard model is not only a model for implementation of expert systems, but also a methodology for decomposing and organizing both the problem-solving knowledge and the problem-solving state. Details of the model are discussed in chapter 4.

1.3.2. Multiple Target Architectures and Knowledge Generalization

Different architectures present different programming models and use different tricks to speed up the computation. Thus the programs need to be restructured on the basis of the knowledge of target architectures in order to utilize the potential parallelism the hardware provides. Generally, there is a trade-off between optimization and portability. Porting a parallel program to another parallel computer of different characteristics requires a major effort which greatly increases the cost of developing software for parallel computers. One solution to this problem is to build multiple targets or retargetable parallel compilers.

Many people believe that the optimization of the machine parallelism needs to be based on the machine-specific instructions that the machine provides; therefore, the heuristics for program optimization are specific to the machine that they are developed for. However, a closer examination reveals that the performance of the parallel architecture is tied to the features of the machine instead of the machine-specific instructions that reflect the features. So ad hoc knowledge needs to be analyzed and converted into knowledge that can be shared by other systems; this requires a model of program transformation that can accommodate such generalization. The feature-directed program optimization model that we discuss in chapter 4 suits this need fairly well. A retargetable parallel compiler not only needs to understand hardware details related to code generation but also needs to have enough knowledge about optimizing parallelism of target machines. This magnifies the difficulties of all the problems in building parallel programming environments.

The major advantage of a multiple target architecture software system is that the knowledge accumulated for each architecture can be transferred to other target machines. Knowledge generalization and transfer can be achieved by analyzing the machine features involved in the knowledge systematically. We discuss this problem in detail in chapter 5.

1.4. Organization of the Thesis

In chapter 2, we discuss some fundamental issues about machine parallelism and program parallelism. These include machine architecture and parallel execution, program dependence, representation and computation of program dependence, abstraction of program parallelism, and programming issues for parallel computers.

In chapter 3, we discuss the foundation of program transformations and survey some restructuring techniques and their applications in parallelism optimization.

In chapter 4, we present a new model for parallel program optimization, called the feature-directed program optimization paradigm. A framework for building optimizing parallel compilers based on the feature-directed program optimization paradigm is derived. A new problem-solving model called the *hier–blackboard* is introduced. This model features opportunistic reasoning with hierarchical organization of the domain knowledge and problem-solving states. We also discuss several approaches for efficient decision-making, knowledge organization, learning, and intelligent user interface, which are the major subjects of the subsequent chapters.

In chapter 5, features of the parallel architecture that affect program optimization are discussed. An object-oriented machine knowledge representation scheme is presented. The utilization of this scheme to build a machine knowledge manipulation system is presented. Problems for collecting, organizing and analyzing machine features and heuristics are also discussed. Some applications of this technology to other related problems are suggested.

In chapter 6, we present a performance prediction model that is designed for parallel compilers. The model is flexible and efficient and can be utilized in parallel compilers to assess the overall performance of a program on a target machine and update the performance estimation incrementally.

In chapter 7, we discuss problems of parallelism matching and techniques to handle the problems for MIMD architectures. Theoretical foundations for array reshaping and message consolidation are discussed. Heuristics for utilizing these and other program transformation techniques are also presented. We also discuss the problem of selecting and fine-tuning pre-optimized algorithms.

In chapter 8, the implementation of an experimental parallel programming environment is discussed and some examples and experimental data are also presented. Finally, in chapter 9, we summarize ideas presented in this thesis, our contributions, and future research directions.

CHAPTER 2
PARALLELISM AND PARALLELISM IMPROVEMENT

2.1. Levels of Parallelism

Parallelism can be exploited at three different levels: the algorithm level, the program level, and the machine level. On each of these levels it can be abstracted as a conceptual concurrency model of computation which is called *virtual machine* of the level.

At the algorithm level, the parallelism lies in the concurrency of the multi-threads of computations allowed by the algorithm. The computational assumptions that the parallel algorithms are based upon represent the virtual machine of the algorithm. Commonly used models include shared-memory models (concurrent-read-exclusive-write, concurrent-read-concurrent-write, etc.) and fixed interconnection networks with constant degree (such as linear array, trees, meshes, hypercubes, systolic arrays, shuffle-exchange, and butterfly networks). *Algorithm level parallelism* can be characterized by the number of virtual processors, the model of concurrent read-writes, the complexity of inter-processor communications, and the complexity class of the parallel execution time on the virtual machine model when expressed as a function of problem size.

At the program level, each parallel programming language defines a virtual machine by the semantics of its parallel control constructs. *Program level parallelism* can be characterized by the control and data dependence constraints imposed by the language and the user's choice of data structures. When an algorithm is represented as a program, its parallelism is limited to the available parallelism of the algorithm under the constraints of the program dependencies.

Machine level parallelism, which is the maximum concurrent execution capacity of the architecture, can be characterized by various machine features such as peak performance, levels of memory hierarchy, ratio of processing and memory access speed, number of pipelines, network topology, and network latency.

When problems are mapped from the algorithm level to the program level or from the program level to the machine level, the differences between the computational models of the two levels may cause parallelism to be lost. For example, when a program is compiled to run on a particular architecture, the effective parallelism of the program is determined by the match between the program and the underlying machine architecture. Similarly, when an algorithm is translated into a program, the concurrent properties of the algorithm may be serialized by the dependence relations inherited from program constructs and data synchronization. In some cases, the concurrency is lost because the limited parallel constructs provided by the programming language simply cannot express the full parallelism of the algorithm.

The utilization of the parallelism of the algorithm on a machine is based on the match of parallelism between the algorithm level and the program level and between the program level and the machine level. At the algorithm-design stage, the user can select the most appropriate algorithm to carry out a particular computation. At the programming stage, the user can select suitable programming languages and data structures to lay down the computation. And then the compiler is supposed to translate the user program into machine language and optimize the program based on its knowledge of the target machine.

Unfortunately, the implementation of algorithms is heavily influenced by the computational model selected, and changing the computational model sometimes requires a complete redesign of the algorithm. There have been many attempts to solve these problems. One approach is to write programs based on a carefully designed programming model. The programming model forms a virtual architecture that executes the programs. Programs written for this programming model can be executed on parallel computers that the virtual architecture is supported. Linda [CaGe88, CaGe89], Express [Para89], Presto [BeLaLe88], ISIS

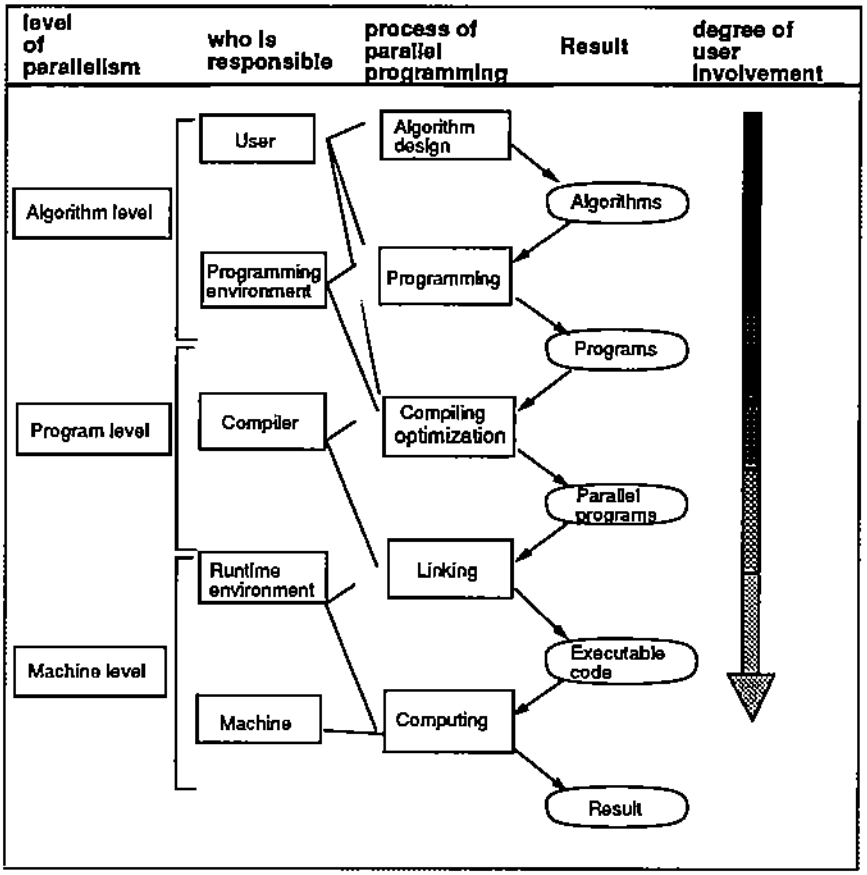

Figure 2.1. *Parallelism and the process of parallel programming.*

[BCJMMKSW90] are just a few examples based on this approach. Another solution, which has achieved mixed results so far, is to allow the user to program the application in high level programming languages that support a limited degree of parallelism (such as Fortran, Fortran 9X, Dino [RoScWe89], Blaze [MeVR87], and Kali [MeVR89]), and relies on compilers or programming environments to utilize the parallelism of the architecture. The user may either express the parallelism explicitly in the program or direct the parallel programming environment or the compiler to map the program to the target machine.

The major difference between these two approaches lies in the level of the abstraction. The former approach usually provides a well-defined, very high level programming model which is not bound to any particular architecture. This model has a smaller application base since some applications may not fit the model well. Another problem is that users have no control over the program's performance which is at the mercy of the implementation of the programming model. Performance may suffer if the target architecture mismatches the computational model. The biggest drawback of this approach is that the huge existing libraries and applications cannot be utilized without complete reprogramming based on the new model; this is very expensive. On the other hand, the compiler approach can utilize the rich existing software libraries by restructuring and recompiling the libraries with parallel compilers or programming environments. Explicit parallelism-programming with general purpose high level languages is usually at a lower level than programming with virtual architectures. Also, capabilities of the compilers and parallel programming environments are still very limited. Retargetable parallel compilers and programming environments that we discuss in this thesis have the potential to automate this process.

2.2. Machine Parallelism

Parallelism can be achieved by processing the data simultaneously. Techniques for achieving simultaneous data processing include:

1. Overlapping I/O and CPU operations.
2. Using multiple independent functional units (FPU, ALU, etc.).
3. Using simultaneous data accesses with multiple memory modules.
4. Using hierarchical memory subsystems (disk, memories, cache memory, registers, etc).
5. Overlaying execution stages of functional units (pipelining).
6. Using multiple processing elements.

These techniques, except for the use of multiple processing elements, have been successfully utilized in most sequential computers. The major reason for the success of these methods is that advanced compiler and operating system techniques make these methods transparent to users. Unfortunately, the same is not true for systems that have multiple processing elements. As we discussed in chapter 1, programming parallel computers with multiple processing elements requires many concepts that do not exist in programming sequential machines.

The major complication of parallel programming lies in the management of the data. When multiple processors can access and update the same data at the same time, simultaneous processing needs to be handled with great care. For example, when several processors modify the same data, the order of the update determines the outcome of the program. Similarly, when multiple copies of the same data exist, the correctness and consistency of the data needs to be of concern also. To preserve the correctness of program execution, the concurrent tasks must be synchronized. Synchronization introduces overhead and causes processors to be idle. Programmers of parallel computers are forced to manage data very carefully since the data communication time can very easily dominate the computation time. Efficient management of data to prevent serialization of the parallelism is another area that needs much more research and is considered by many experts as a bottleneck of parallel programming.

Another problem is that computation must be split into parts that can be run simultaneously on the processors. This includes two kinds of parallelism: *multiprogramming* and *multi-tasking*. In multiprogramming, several programs are allowed to run concurrently on the parallel computer. The resources of the systems are shared by the programs, but there is no further interaction between the programs. In contrast, multi-tasking splits a single program into multiple parts that are called tasks to run on a parallel computer. The tasks coordinate with each other to compute the solution of the problem. The parallelism is exploited by running the tasks on multiple processors concurrently.

To achieve the maximum speed up, the workload of each processor must be evenly divided. This problem is called *load balancing*. Since processes in multiprogramming are independent, the load balancing problem in multiprogramming usually concerns the utilization of the resources and the throughput of the system. The scheduling of the multiprogramming is usually decided dynamically at the runtime by the operating system of the target machine. On the other hand, tasks in the multi-tasking environment are usually related, and the tasks and the relationship between them need to be analyzed to achieve a balancing load. Therefore, multi-tasking is usually scheduled statically at compile time, although some decisions may be deferred to the run time.

2.2.1. Variations of Parallel Computers

Parallel computers can be classified by different criteria. One of the most widely accepted classifications was based on the number of instruction streams and data streams [Flynn66]. An *instruction stream* is a sequence of instructions as executed by the machine and a *data stream* is a sequence of data as referenced by the instruction stream. Flynn classified computer architectures into four different classes:

- SISD: Single instruction stream - single data stream
- SIMD: Single instruction stream - multiple data streams
- MISD: multiple instruction streams - single data stream
- MIMD: multiple instruction streams - multiple data streams

In SISD systems, instructions are executed sequentially. Overlapping of instructions is possible (such as overlapping data fetching and processing or overlapping execution stages) in order to speed up the computation. Even though an SISD may have more than one functional unit (such as floating point unit and integer unit), all of them are under the supervision of one control unit.

An SIMD machine contains multiple processing units which operate synchronously in lock step under the same control units. Most SIMD machines are special purpose array processors that are applied mainly to signal and image processing. Examples of machines in this category include IBM GF11 [BeDeWe85], Connection Machine [Hillis85], and FPS 164/MAX.

The MISD model is considered impractical and no research or commercial effort has ever been placed in this category.

The MIMD architecture consists of a set of sequential processing elements (PEs), each of which may execute instructions independent of each other. A generic MIMD machine consists of a set of PEs, a set of memory modules and an interconnection network that interconnects PEs, memory modules and peripheral devices.

The most important property of MIMD computers is the memory hierarchy and the communication mechanism. This is because in most systems, especially machines that have several levels of memory hierarchy, misuse of fast memory or the communication network introduces unwanted overhead and can significantly degrade the performance of the system.

2.2.1.1. Network Topologies

Processing elements, memory modules and peripheral devices in an MIMD computer are connected to each other through the interconnection network. Interconnection can be dynamic or static. In a dynamic interconnection network, switches are provided so that elements in the system can be connected under program control. Crossbar switches, shuffle exchange networks, etc. are widely used dynamic interconnection networks. CHiP computers [KWGCS84] are highly configurable MIMD machines in which different topologies of the interconnection can be formed under the program control of the switches. The static interconnection has a fixed topology by which processing elements are arranged in multidimensional patterns and permanently connected to each other. The most widely used interconnection method is the bus structure. In this method, all elements share a single connection medium and employ a variety of techniques such as token passing or time-sliced broadcasting to assure correct data transmission. Different types of communication networks include mesh, hypercube, torus, cubic connected, and butterfly.

2.2.1.2. Memory Hierarchy

Machines that use distributed local memories attached to processing elements are called *distributed-memory MIMD systems*. Message passing is the communication method among the processing elements in distributed-memory MIMD systems. Examples of distributed-memory MIMD systems include nCUBE 2, Intel iPSC/2, Intel Touchstone, and Pringle.

Shared-memory MIMD machines are tightly coupled systems using shared memory among multiple processors. Shared-memory MIMD machines can have multiple levels of memory hierarchy. In the extreme, a machine with complete memory hierarchy may include global memory, cluster memories, local memories, cache memories, or even multiple levels of cache memories [KDLS86]. Shared memory MIMD machines are normally bus connected or direct connected. Examples of shared memory MIMD machines include Sequent Symmetry, Alliant FX/80, and Cray X-MP.

2.2.2. Programming Issues for MIMD Parallel Computers

For both shared and non-shared machines, data management presents the biggest challenge to the utilization of the parallel architectures. Bad memory management may cause network or bus contention and dramatically degrade the performance of the system. For shared-memory systems, the major programming issues include task creation and scheduling data allocation, locality, memory and cache management, and minimization of synchronization. For distributed-memory MIMD machines, process creation, data decomposition, load balancing, locality, message routing, and deadlock prevention are the most significant problems. Distributed-memory parallel computers are normally more difficult to program because data communication needs to be done through explicit message passing, and the communication cost is normally much higher than that of its shared memory counterparts.

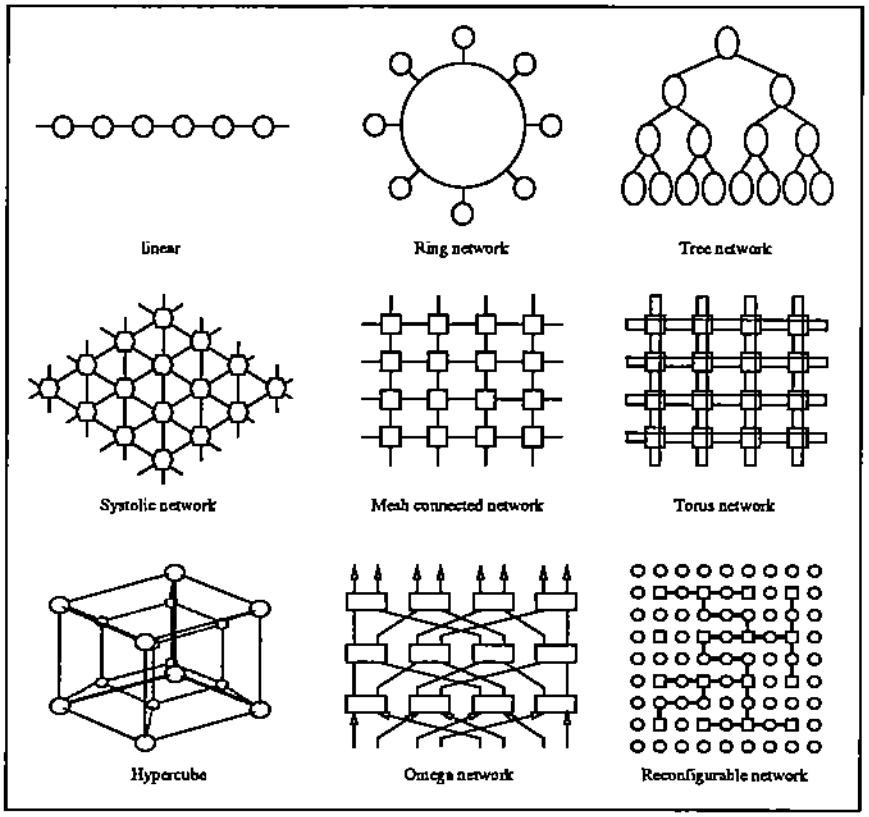

Figure 2.2. *Example of different network topologies.*

2.2.3. Computational Models

The computer system as perceived by users may be quite different from the physical device that carries out the computations. This is because the functionalities and limitations of the machine may be modified, limited, or extended by languages, operating system, software, microcode, or optional hardware of the system. In other words, the computational model of the computer can be viewed as a hierarchy of layers that includes hardware devices, operating system, programming languages, software environments, etc. Each level of the "machine" that the users see appears to be different and possesses different functionalities and the methodologies for using the machine. When the user is using the computer, the actual computing device that "appears" to the user depends on which level of the hierarchy the user is at. We called these virtual machines the *computational models on the machine.*

Based on the level of abstraction, the computational model can be viewed as a hierarchy of levels (figure 2.5), with the physical hardware at the lowest level. The operating system and utilities are built on top of the physical hardware level, and the programming languages are implemented on top of both the operating system and the hardware. The applications and the software systems are the topmost layer in the hierarchy, which usually is what users see as the computer. This division is an extension of the division of machine, program, and algorithm level of parallelism in section 2.1 and can be further extended to divide each level into sublevels. The computational models of the levels are composed of the features and methodologies of using the levels and they form the interface between the levels. The upper levels can be viewed as programmed on the lower levels. Starting from the bottom up, the computational model of the hardware level consists of specifications and features of the hardware (for example, the number of vector registers, and cost ratio of memory access and processing). The operating system level includes operations and control for multiprogramming and utilization of the devices (for example, process start up time, swapping time, and self-scheduling operations). The programming language level is defined by the constructs and concepts of the programming language. It defines data structures, instructions and control structures to control the execution of the program. The programming language level forms the foundation for the program parallelism. The application level consists of various applications and is out of the scope of our interest.

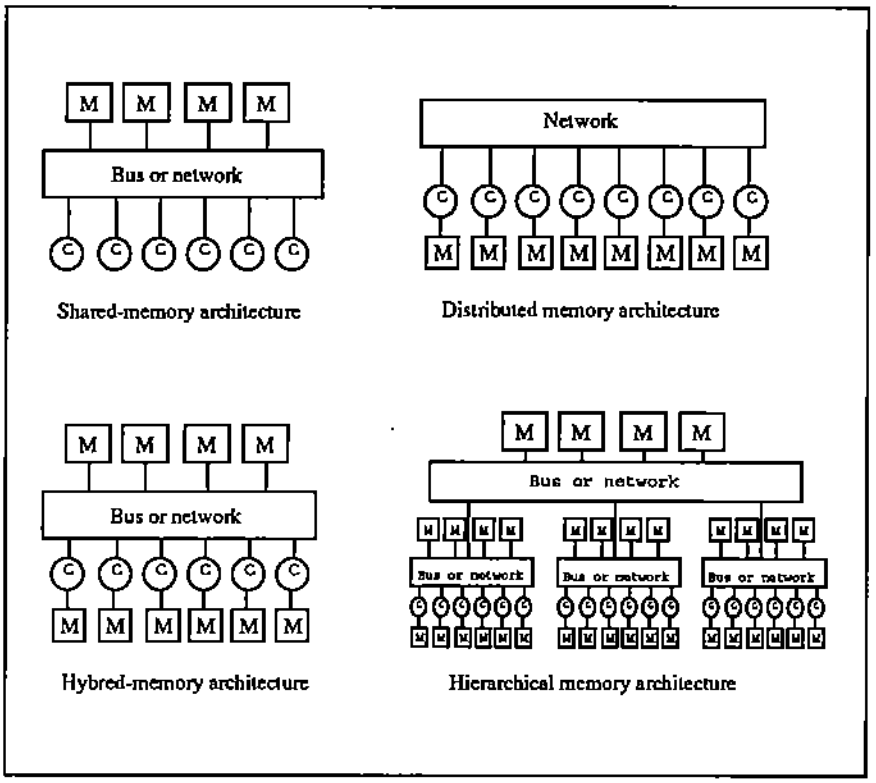

Figure 2.3. *Different types of MIMD computers.*

Features of a level may be preserved, obstructed or modified as we go across the level boundaries. Some new features may be the abstraction of the features of the lower levels. Some may be modified because of the limitations of the upper levels. When the machine is seen from a level down, the actual characteristics of the computational model consist of the features that belong to the current level and those of lower levels that are not obstructed by the level boundaries.

Adopting the notion of computational models at different levels of the system provides a good abstraction of the levels. The model can shield the users of the level from the lower levels and is therefore conceptually easier to manipulate. It also makes identifying differences between levels easier.

2.3. Program Parallelism

2.3.1. Granularity of the Program Parallelism

Program level parallelism can be further divided into three concurrency levels: *task, micro task,* and *operation.* At the task level, a program is decomposed into processes which may be run on different processors. At the operation level, vector or scalar operations are the units of computation. The size of the vector operation represents the degree of concurrency at this level. The micro task level is the level between the task level and the operation level and is often characterized by loop bodies. More precisely, inside a task, operations are grouped into micro tasks, which are blocks of code that are executed between synchronization points.

Efficient utilization of a program on a parallel machine depends on the composition of tasks and balance between operation level parallelism and task level parallelism. A parallel compiler will have to decide the best way to form the tasks, vector operations, and the scheduling of the tasks. Decisions about the granularity of the tasks, structure of the tasks and the scheduling of the tasks are based on many factors; a non-exhaustive list includes the cost of task dispatching, distribution of shared data, compile and run time knowledge of the program and system, and ratios of computation and external data references of the tasks. These decisions can be made either statically at compile time or dynamically at run time. Making the decisions at run time has the advantage of having additional information (such as values of some variables) that is not available at compile time. However, the negative side is that it also increases the execution time significantly. Therefore, a

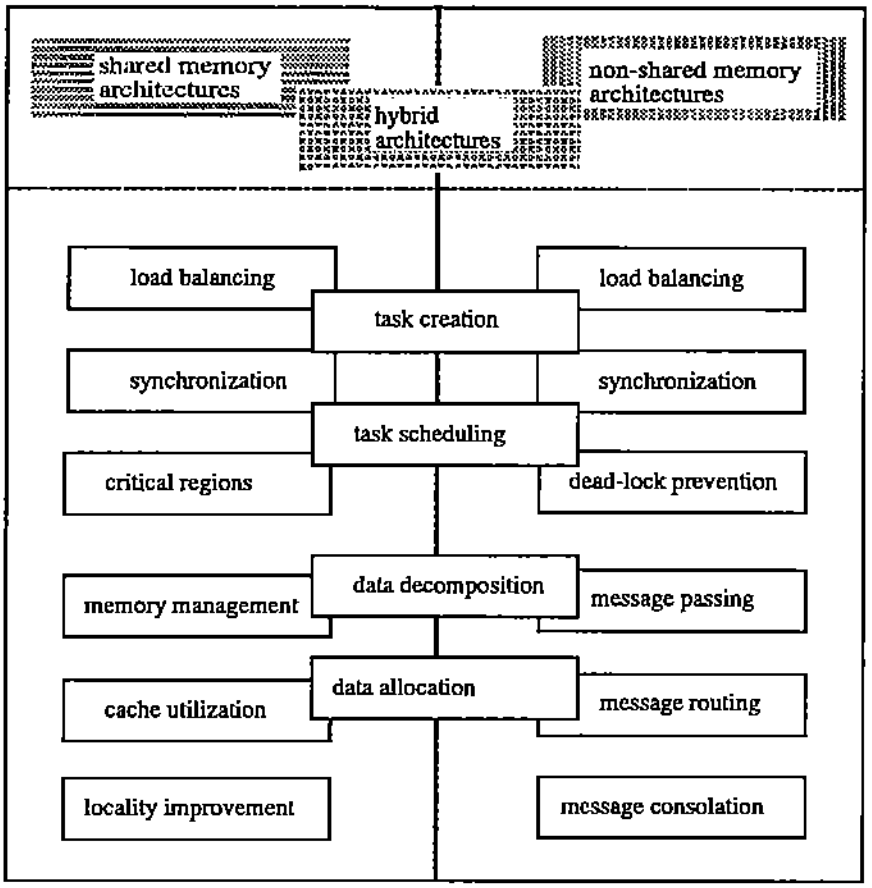

Figure 2.4. *Programming issues for MIMD computers.*

compromise between compile and run time decision making is needed.

2.3.2. Program Dependence

Parallel execution of the programs is governed by the dependence relations of the program. There are two types of program dependence: control dependence and data dependence. The control dependence [FeOtWa83] specifies the preconditions on the operations which are required for them to be actually executed (such as loops, conditions, and exit statements). The data dependence represents the set of essential constraints on the execution order of the data references. Together, these two types of dependencies form a complete summary of the semantics of the program.

Since the program dependence specifies the constraints that the program parallelization process must respect, violating the dependence relation may cause data access and modifications to happen in the wrong order and thus change the meaning of the program.

There are four types of data dependence relations: *input dependence, flow dependence, anti-dependence* and *output dependence* [KKLPW81] and [Wolfe82]. In the following discussion, a *program component* can be a data reference, an expression, an assignment statement or a compound statement such as a loop or conditional statement.

Definition 2.1 Consider two, not necessarily distinct, components S_1 and S_2; we say that:

1. There is a flow dependence from S_1 to S_2, if a value computed by S_1 is stored in a location associated with some variable name x which is later referenced in S_2 before other components overwrite the value. We denote the flow dependence relation as $\delta_x : S_1 \rightarrow S_2$.
2. There is an anti-dependence from S_1 to S_2, if a variable x referenced by S_1 must be used before it is overwritten by S_2 and is denoted by $\hat{\delta}_x : S_1 \rightarrow S_2$.

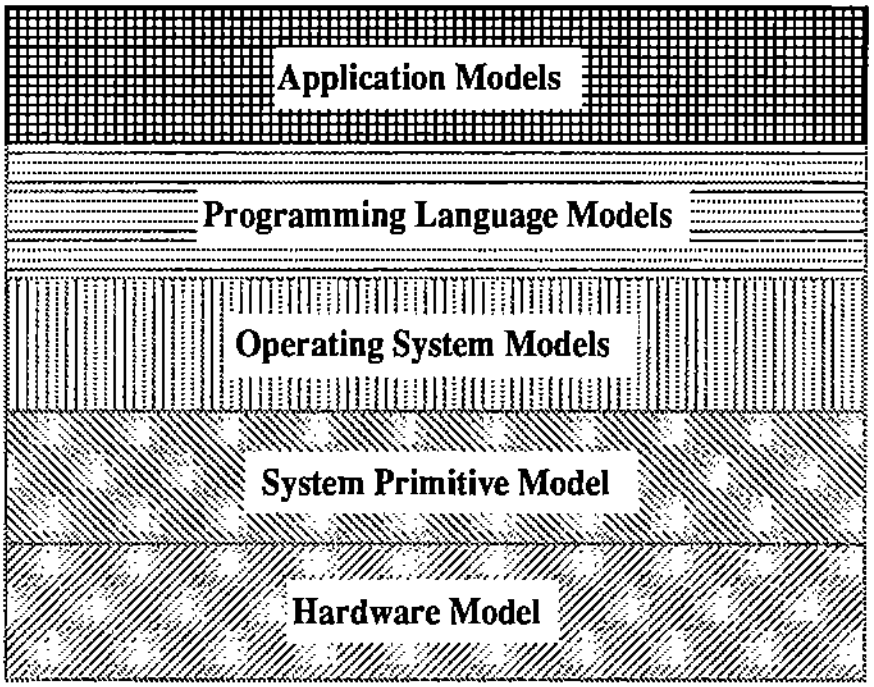

Figure 2.5. *A hierarchical view of the computational models.*

3. There is an output dependence from S_1 to S_2, if a value computed by S_1 is stored in a location associated with some variable x and is later overwritten by computations in S_2. This relationship is denoted by $\delta_x^o : S_1 \rightarrow S_2$.
4. There is an input dependence from S_1 to S_2, if both components use the same value stored in a location associated with variable x and is denoted by $\delta_x^I : S_1 \rightarrow S_2$.

These data dependencies impose constraints on the execution order of the components. If there are flow, anti- or output dependencies from S_1 to S_2, component S_1 must be executed before component S_2. These constraints are necessary in order to preserve the input-output semantic of the program. Note that the input dependence is a reflexive relation and does not impose any constraint on the potential parallel execution of the two components. However, it can be used to compute the cache-miss ratios for memory management [GaJaGa87].

A loop instance in a loop or multiple loops can be identified by the index or indices associated with the instance. A dependence relation that crosses from one loop instance into another is said to have a distance and the distance is defined to be the difference of the loop indices.

Definition 2.2 Suppose there is a dependence relation from statement S_1 in loop index tuple I_1 to statement S_2 in loop index tuple I_2, then the *distance vector* of the dependence is defined to be $I_2 - I_1$

Not all optimization techniques need the detailed knowledge about the distances between references of the dependence. In some cases, only the signs of the vector elements are needed. The *data dependence direction vector* is defined as the vector of the signs of the distance vector of a dependence. For the representation of the direction vector, '<', '=', '>' are used to represent the relationship between the indices of the source and destination array elements, where '<' means the dependence across a loop boundary in a forward direction; '>' means the dependence across a loop boundary in a backward direction; '=' means that the dependence appears in the same loop instance.

Algorithms to compute the program dependence graph can be found in [Wolfe82, AlKe84a, Wang85b, BuCy86, Bane88, Wolfe89, Zima90]. Interprocedural dependence analysis can be found in [Allen74, Bane76, Hecht77, Barth78, GrWe76, KaUl76, Myer81, CoKe88, LiYe88a, LiYe88b].

In figure 2.6, we show the program dependence graph of the following sample program. The labels of the dependence are the direction vectors of the dependence.

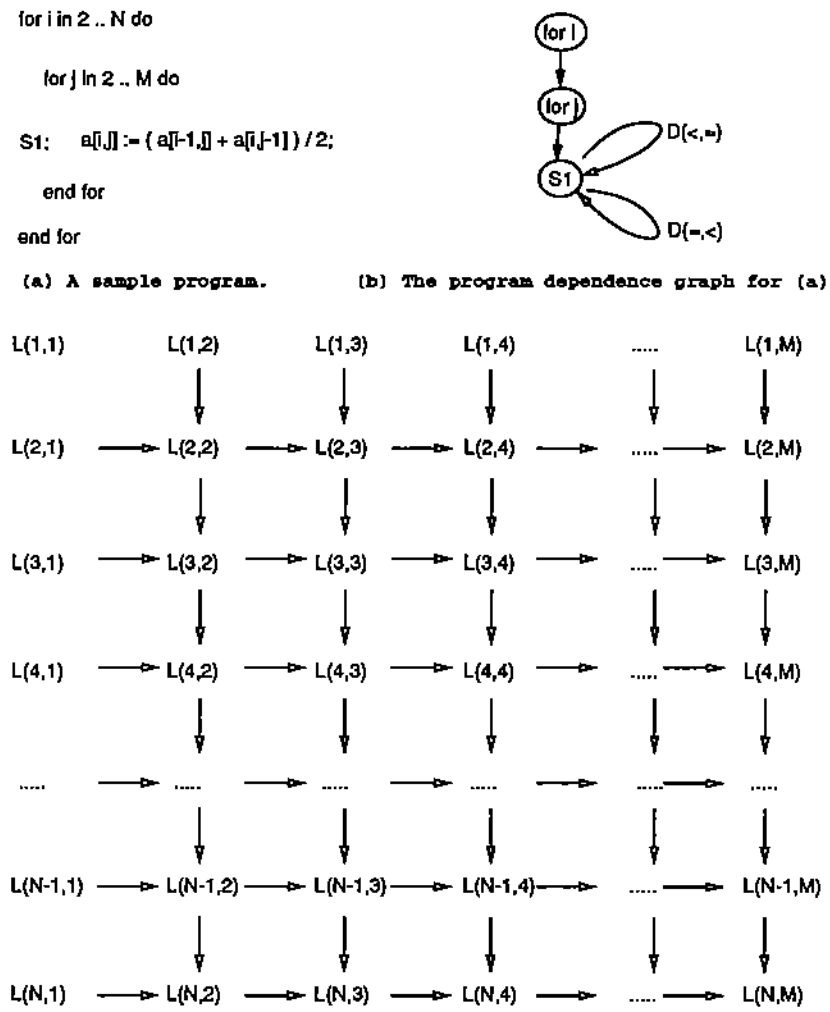

Figure 2.6. *A sample Blaze program, its dependence graph and expanded dependence.*

2.3.2.1. Representation of Program Dependence Graph

There are two kinds of dependencies in the program dependence graph (that is generated by the dependence decision algorithm): those that are proven to exist ("proven dependencies") and those that are assumed to exist ("assumed dependencies"). The "assumed dependencies" are included because the decision algorithm fails to prove the non-existence of the dependencies. For example, most compilers assume the existence of the dependence when they encounter one of the following situations: a subscript of an array reference is non-linear with respect to loop indices, a subscript contains variables, or there are variables in loop bounds. Existing compilers make no distinction between "proven dependencies" and "assumed dependencies." Subsequently, the generated dependence graph contains many dependence relations that do not exist in the program. These "false dependencies" may prevent the compiler from performing certain transformations or may even sequentialize the program. This problem may be solved by generating run-time tests to validate the existence of the dependencies at the run-time. However, run-time tests are expensive, and when dependencies are generated at a different phase, tests that have been done in the dependence analysis are often repeated at the run time. We solve this problem by introducing a generalized representation for the program dependence graph. In our representation scheme, the type of a data dependence is stored with the dependence. For an "assumed dependence," conditions for it to exist are also stored. Also, a confidence factor is assigned to each dependence. The first information can be used to minimize the run-time tests when such tests are warranted. The confidence factor can help the inference engine of the optimizer to make better decisions. Both types of

information can be provided by a modified dependence decision algorithm. Other information can also be stored, and this set of information is collectively called the attributes of the dependence.

2.4. Optimization of Parallelism with Program Transformations

2.4.1. Abstraction of Program Parallelism Based on Program Dependence Graph

Abstracting parallelism out of a program is the first step toward utilization of the program parallelism. For compilers that allow assertions or interactive user involvement, program parallelism recognition is largely the responsibility of the programmer. For automatic parallelism optimizing compilers, the compilers will have to discover the potential parallelism by examining the program. The program dependence graph represents the constraints on the parallel execution of the program. The compiler usually realizes the parallelism of the program by verifying that certain patterns of the dependence do not exist in the program. For example, the following *vectorization test* can be used to determine whether a loop can be vectorized.

Example 2.1 Vectorization test. Let $PG(B,D)$ be a program dependence graph of a loop L with index i. If there is a dependence arc from node S_1 to node S_2 whose direction for index i is '<' and S_1 precedes S_2 in the topological sort order or $S_1 = S_2$, then the loop is not vectorizable.

Techniques that can be applied to abstract the parallelism of the program include pattern recognition, graph-reduction, and heuristics.

The following Prolog procedure shows how a simple pattern recognition technique can be applied to discover the fact that the program fragment is an inner product operation.

```
/* Stmt in loop Loop is an inner-product of vector A1 and A2 */
innerProduct(Loop, Stmt, A1, A2) :-
        inner_most_loop(Loop),
        loopIndex(Loop, I),
        aStatementIn(Loop, Stmt),
        loop_distributable(Loop, Stmt),
        accumulateOver(Stmt, '+', Var, Expr),
        expression(Expr, '*', A1, A2),
        simpleIndexSubscript(A1, I),
        simpleIndexSubscript(A2, I).
```

What the above procedure shows is that the pattern of an inner-product is a statement inside a loop whose index is I, and the loop can be distributed into the statements if there is more than one statement in the loop. The computation is to multiply elements of two arrays and the results are accumulated into the variable on the left hand side of the statement. Furthermore, the subscripts of the arrays that contain index I are simple subscripts (contain only variable I).

The pattern defined by the above program is rather general. For instance, all program fragments in figure 2.7 except figure 2.7(e) satisfy this pattern.

The program in figure 2.7(e) fails the inner-product test because the accumulation of the pair-wise multiplications is in the outer loop instead of the inner loop. By interchanging the execution order of the loop, the program can be converted into inner-product operations. This situation can be covered by the following rule which says that if the statement is a nested loop and the loop that the accumulation is based on can be moved to be the innermost, then the program contains an inner-product.

```
(a)  for i in 1 .. n do
        x:= x + a[i] * b[i];
(b)  for i in 1 .. n do
        for j in 1 .. m do
              y[i] := y[i] + a[i,j] * x[j];
(c)  for i in 1 .. n do
        for j in 1 .. m do
              for k in 1 .. l do
                    c[i,j] := c[i,j] + a[i,k] * b[k,j];
(d)  for i in 1 .. n do
        for j in 1 .. m do
              b[i,j] := a[i,j] ** 2;
              y[i] := y[i] + a[i,j] * x[j];
              c[i] := x[j]+y[i];
        endfor
(e)  for j in 1 .. m do
        for i in 1 .. n do
              y[i] := y[i] + a[i,j] * x[j];
```

Figure 2.7. *Example of program fragments that contain inner-products.*

```
generalInnerProduct(NestedLoops, Loop, Stmt, A1, A2) :-
      interchangableLoopOrder(Nested, NewOrder),
      innermostLoop(NewOrder, Loop),
      innerProduct(Loop, Stmt, A1, A2).
```

After the potential inner-product loops and statements are identified, the statements can be split from the loops and translated into inner-product statements. Figure 2.8 shows the resulting programs by translating the computation into explicit inner-product operations. Note that in (a), (b), and (c), the programs are simply translated into inner-products. But in (d), the statement $y[i] := y[i] + a[i,j]*x[j]$; is first swapped with the statement $b[i,j] := a[i,j]**2$; to separate it from the rest of the loop; then the j loop is distributed into the two blocks of statements and the first generated j loop is translated into inner-products. And (e) is obtained by first interchanging the loops to move the jth loop to the innermost and then translate it into inner-product.

```
(a)  x:= innerProduct(a, b, 1, n)
(b)  for i in 1 .. n do
        y[i] := innerProduct(a[i,*], x[*], 1, m);
(c)  for i in 1 .. n do
        for j in 1 .. m do
              c[i,j] := innerProduct(a[i,*],b[*,j], 1, l);
(d)  for i in 1 .. n do
        y[i] := innerProduct(a[i,*], x[*], 1, m);
        for j in 1 .. m do
              b[i,j] := a[i,j] ** 2;
              c[i] := x[j]+y[i];
        endfor
      endfor
(e)  for i in 1 .. n do
        y[i] := innerProduct(a[i,*], x[*], 1, m);
```

Figure 2.8. *Result of translating the statements into inner-products.*

The techniques that we have just described to translate the program fragments into inner-product operations are called program transformations. Another technique to improve the parallelism of the program is to substitute the program fragment with a previously optimized program for the same target machine. These two approaches will be discussed in more detail in the next section.

2.4.2. Optimization of Program Parallelism

The program-restructuring process changes a program into a semantically equivalent representation of the program. The purpose of program restructuring is to modify the structure of the program to utilize the parallelism potential of the target machine.

Definition 3.1 A program is said to be *semantically equivalent* to another program if their input-output behaviors are the same.

Definition 3.2 The *potential parallelism of a program on a target machine* is defined to be the best parallel execution order of the program on the target machine that is semantically equivalent to the original program.

Definition 3.3 The *actual parallelism of a program on a target machine* is the degree of parallelism that the current form of the program (i.e. with no modifications to the structure of the program) can achieve on the target machine. In other words, the actual parallelism is a measure of the match between the current form of the program and the target machine. The major factors in deciding the *actual parallelism of a program on the target machine* are data and control dependencies and patterns of data references.

The goal of the parallelism optimization process is to find a semantically equivalent program so that its *actual parallelism* on the target machine is as close to the *potential parallelism* of the original program as possible.

2.4.2.1. Parallelism Realization and Program Restructuring

The process of optimizing program parallelism consists of two steps: the program-restructuring process and the program-realization process. The program-restructuring process improves the potential parallelism of the program by modifying the structure of the internal program representation. The program-realization process maps the program onto the computational model of the target machine by effectively utilizing the concurrency potential of the machine. The program-restructuring process optimizes the potential parallelism of the program for a target machine, while the program-realization process actually maps the program onto the target machine to utilize the actual parallelism of the program on the target machine.

Based on the dependence constraints of the program and the feature descriptions of the target machine, the program-realization process partitions the program into operation blocks and composes them to form vector operations, micro tasks, tasks, and processes. In the abstract, the program-realization process can be viewed as a function:

$$Program\_realization: Computational\_model\ X\ Programs \rightarrow Program_C$$

where *Programs* are program dependence graphs that are semantically equivalent to the original program, *Computational_model* is the computational model of the target architecture (note that target architecture here is an abstract concept which does not need to have a physical implementation in hardware), and the elements of $Program_C$ are program dependence graphs that are annotated with parallelism and run-time information such as processor assignments, synchronization, and vectorizable or parallelizable loops.

Strictly speaking, the program-realization process does not improve the true parallelism of the program. It simply takes the current form of the computation, as represented by the program dependence graph, and applies a mapping to convert the program into a parallel form. For example, for multiprocessor systems the outermost parallelizable loop can be used to generate tasks. For machines with vector capability, the innermost loop can be vectorized (if it is legal to do so). The synchronization technique that is provided by the computational model is used to satisfy any data dependence not already satisfied by sequential execution of parts of the program. More sophisticated modification of the program (such as interchanging the nested loops or even non-nested loops) is the task of program restructuring. Many existing compilers for parallel architectures provide program parallelism-realization capability but not necessarily the program parallelism-optimization capability.

2.4.2.2. Two Approaches for Program Parallelism Optimization

The program-restructuring process maps a program into a functionally equivalent program by altering the control or data structures of the program. The goal of the restructuring is to properly change the order, the decomposition, and the allocation of the data and control structures of the program so that the achievable parallelism can approach the potential parallelism of the program on the target machine.

There are two different approaches to restructuring the programs: the *pre-optimized algorithm substitution* approach and the *program transformation* approach. The *pre-optimized algorithm substitution* approach replaces the program fragment under consideration with an algorithm in the library that has been pre-optimized for the particular target machine. The precondition for this substitution is that the functionalities of the program fragment need to match those of the pre-optimized algorithm. The *program transformation* approach improves the match between the program level parallelism and the machine level parallelism in a stepwise refinement fashion. The program transformation approach breaks the program restructuring process into basic steps that are called *program transformations*. Each program transformation alters the structure of a program segment in order to improve the parallelism of the program. It involves techniques such as changing the instruction execution order (by forward substitutions, statement reordering, etc), modifying program control (by loop interchange, loop distribution, etc), and eliminating unnecessary data accesses and modification (by data localization, block transfer, cache optimization, dead code elimination, etc).

Since a transformation only modifies the program structure slightly, it is possible to define the conditions which the program must satisfy so that the resulting program will have the same input-output behavior as the original program. These conditions are called the *preconditions* of the transformation. If a program satisfies the preconditions of a transformation, the transformation is said to be *applicable* to the program.

Optimization of a program requires careful selection of a sequence of applicable transformations based on the understanding of the program and the target machine. The composition function of the transformations maps the original program into the final form and determines the net effect of the complete sequence of the transformations.

2.4.2.3. Pre-optimized Algorithm Substitution Versus Program Transformation

The major difference between the pre-optimized algorithm substitution and program transformation approaches lies in the granularity of the changes. Program transformation modifies the program structure step by step, whereas the pre-optimized algorithm substitution takes the "wholesale" approach by replacing a whole program fragment by a pre-optimized algorithm in the library.

The major problem with the pre-optimized algorithm substitution lies in the difficulty of recognizing the opportunity for algorithm substitution. Comparing the semantics of two programs is an NP-complete problem and it is usually very difficult if not impossible for a software system to match two programs. This approach is used by some parallel programming environments since the programmer can provide the needed information that a software system cannot obtain. Another drawback of this approach is that in most systems only a small set of pre-optimized algorithms is available. This is due to the high cost of constructing the library of pre-optimized algorithms which limits the scope of the application also. On the other hand, once the opportunity for algorithm substitution is recognized, this approach can usually achieve very good results since special attention has been paid to the efficiency of the pre-optimized algorithms. The FAUST project [GGJMG89] at CSRD, University of Illinois takes the expert systems approach to choose the appropriate algorithms that match the user program.

Optimizing parallel compilers usually take the program transformation approach, since compilers are particularly well-suited for mechanical analysis of the program dependence graph and for verifying the preconditions of the transformations. However, the problem with this approach is that the decision for restructuring the program structure is fragmented. The effect of a transformation on the utilization of the parallelism may not be clear at the time when the transformation is selected. Global analysis of the transformation is needed. Selecting a right sequence of transformations is highly target architecture and program dependent and is very difficult. Finding a generic framework for selecting transformations is even more difficult and most experts rely on heuristics. Another problem in building parallel compilers is that most studies of the transformations center on the theories about the pre-conditions of the transformations and algorithms for carrying out the transformations. Few efforts have studied the practical problems of program transformations such as recognizing the opportunities for applying the transformations or methodologies for selecting the best transformation among several applicable transformations. This may be the main reason that, despite the development of so many different program transformation techniques during the last decade, the performance of parallel compilers is still disappointing.

In this research, we take the following stance concerning the two different approaches:

- The two approaches are not necessarily mutually exclusive and they should be applied together to overcome the shortcomings of each other.

- At the compiler level, the pre-optimized algorithm substitution can be applied to some well-defined problems, such as those functions defined in BLAS[I,II,III]. More complicated algorithm substitution is possible, but recognizing the opportunity for application is usually too difficult for the compiler to handle. For programming environments, this requirement is less critical since the user can provide some information to ease the difficulty the environment faces in recognizing the chance for applying pre-optimized algorithms.
- If the opportunity for algorithm substitution is found, it should take precedence over program transformation.
- It is impractical to optimize every useful algorithm for all different target machines. Therefore, for similar architectures, we only need to pre-optimize an algorithm for a particular type of machine and equip the system with rules to select the pre-optimized algorithm that is optimized for a similar machine and apply a set of program transformations to fine-tune the pre-optimized algorithm to fit the actual target machine. This approach will be discussed in more detail Chapter 7.

2.4.3. Classifications of Program Transformation Techniques

Program transformation techniques change the structure of the program by modifying control dependence(s), data dependence(s), the decomposition and allocation of instructions and data, data reference patterns, or the combinations of above. Program transformation techniques can be classified into two categories: machine-independent and machine-dependent transformations. This classification is not disjoint and the same transformation may be included in both categories for different objectives.

The machine-independent program transformations are used to improve the general parallelism of the program such as breaking dependence cycles, improving locality, and eliminating redundant instructions. More specifically, transformations that can be used to break dependence cycles include scalar expansion, variable renaming, statement splitting, and forward substitution. Transformations to improve locality include statement reordering, array gathering, subscript blocking, array reshaping, and loop interchanging. Due to the difference in architectural requirements, it may be wise to delay the locality improvement until the target machine is chosen. However, some obvious improvement can still be achieved without knowing the target machine since locality is almost always good for parallel execution. Transformations to eliminate redundant instructions include code motion, dead-code elimination, and loop merging. Most traditional compiler optimization techniques to remove redundant instructions can also be applied.

Machine-dependent transformations make use of the special knowledge about the underlying parallel architecture to improve the match between the program and the target machine. Machine-dependent transformations can be used to improve vectorization and parallelization, decrease synchronization time, minimize data access time, schedule execution, distribute data, etc. Transformations to improve vectorization include loop distribution, loop merging, loop blocking, loop unrolling, loop collapsing, loop interchanging, loop spreading, Boolean recognition, scalar expansion, if matching, and if removal. Transformations to improve parallelization include loop interchanging, loop blocking, loop merging, loop distribution, loop collapsing, loop coalescing, vector scalarizing, code motion, high level loop spreading, low level operation spreading, and doacross scheduling. Transformations to do memory optimization and minimize synchronization include array decomposition and distribution, array copying, synchronization, cache utilization, array reshaping, array block transfer, array gathering, array scattering, array scalarizing, and scalar expansion. Transformations to create concurrent tasks include loop interchange, loop blocking, and cyclic loop blocking. Transformations to schedule execution include high level loop spreading, doacross scheduling, self-scheduling, and run-time scheduling. This classification is summarized in the figure 2.9 below. A survey of many program transformation techniques can be found in [Zima90].

Note that program transformations are merely mechanical techniques to change the structure of the program. Inappropriate application of the transformations is likely to cause more harm than good. To have a positive effect on performance, the transformations must be carefully chosen based on clear objectives and well-thought out heuristics to take full advantage of the parallelism provided by the target architecture.

The capability of a compiler to optimize the program parallelism is usually determined by the richness and effectiveness of the heuristics that the compiler used. Due to the lack of systematic research in this area, there is no readily available resource of heuristics for parallel compiler writers to employ. Consequently, only scattered heuristics are utilized in most parallel compilers. What is needed is to integrate the heuristics of applying the transformations into the knowledge database so that the inference engine of the compiler can

utilize them systematically. A few dozen well-known heuristics are described in [Allen83, Wolfe82, WaGa89]. Some new heuristics for utilizing these transformations to optimize the programs on different classes of target machines are presented in chapters 7 and 8.

2.5. Summary

In this chapter, we discussed some fundamental problems about parallel programming. We first discussed levels of parallelism and how computations are mapped from one level to another. We described some basic concepts of machine parallelism and computational models. We also discussed program-dependence relations and issues in representing the program dependence graph.

We talked about the abstraction of program parallelism and give some simple examples. We then discussed two different ways to improve program parallelism and some theoretic foundations for program transformation. In the next chapter we will discuss a framework for integrating heuristics, machine features, and performance prediction into the decision making of the program optimization process.

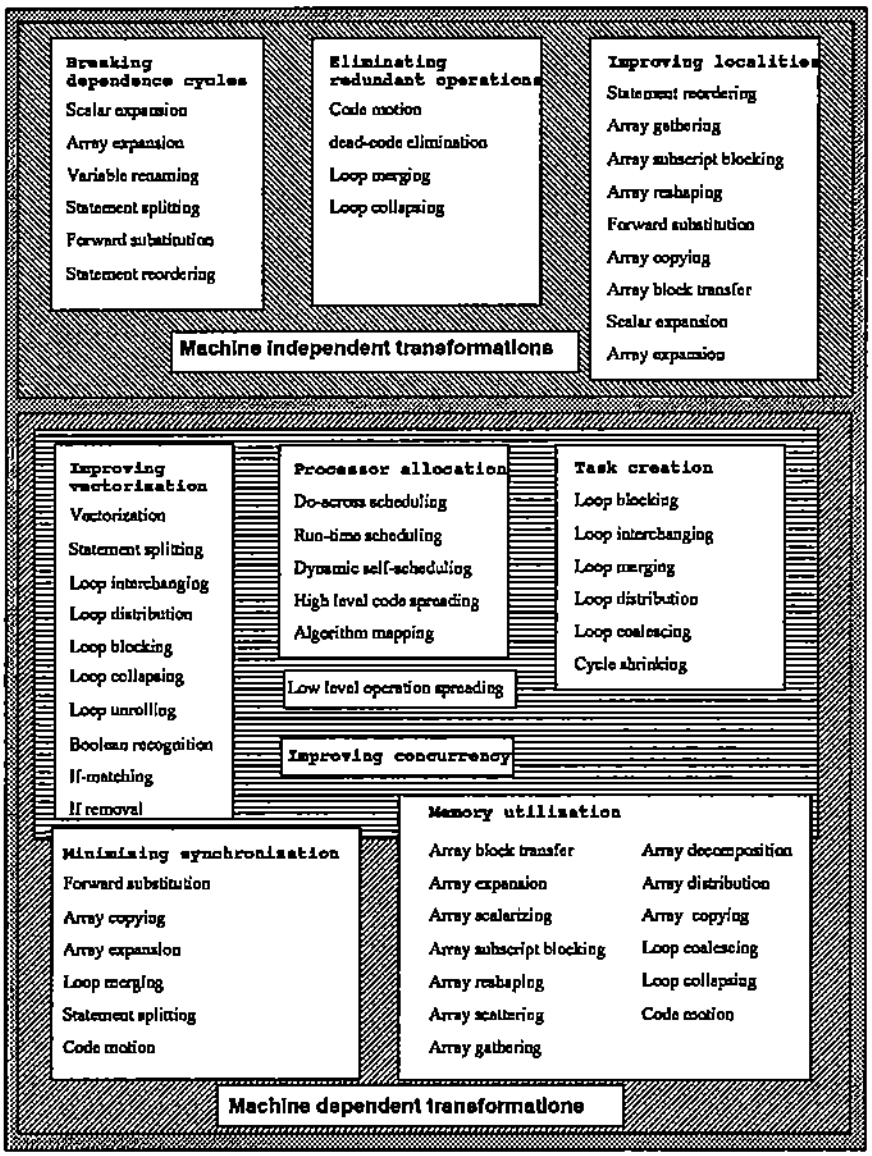

Figure 2.9. *Classification of program transformation techniques based on objectives.*

CHAPTER 3
TOWARD INTELLIGENT PARALLEL COMPILERS

3.1. Need for Intelligence in Parallel Compilers

For a compiler to generate reasonable parallel code for a parallel architecture, it has to be able to recognize the parallelism of the hardware architecture, abstract the parallelism available in the user program, and use suitable algorithms or heuristics to match the program parallelism with the machine parallelism. These tasks require a fairly high level of intelligence. Unfortunately, most parallel compilers today lack such needed intelligence. Although a few existing parallel compilers exhibit limited degrees of intelligence, none is capable of generating acceptable code for parallel computers without extensive user guidance.

Below we will use two simple examples to demonstrate these difficulties. The first example involves parallelizing a matrix-vector product program which nicely illustrates the complexity of the problem because very few dependence relations are involved and many transformations are possible. The program is a simple nested iteration:

```
for i in [1 .. n] do
    for j in [1 .. m] do
        y[i] = y[i] + a[i,j]*x[j];
    end for;
end for;
```

We assume that the resulting vector y has been previously initialized to zero. We seek to transform this program for three different target architectures: the BBN butterfly, the Alliant FX/8, and the Pringle. These three parallel computers support different views of parallelism and represent a wide variety of parallel architectures.

The BBN Butterfly supports a model of computation based on a synchronized access to shared data in a global address space. The memory modules and processors are connected to a multiple stage network. The cost ratio of global and local memory accesses is about two to one.

The Alliant FX/8 is an eight processor shared memory system where each processor has a vector pipe. The processors are connected by a crossbar switch to a shared cache, which, in term, is connected to a shared memory through a high speed bus. Lightweight threads are supported for programmer-directed task scheduling.

The Pringle† [KGSF84, KWGCS84] is a 64 processor, non-shared memory CHiP prototype system. The Pringle demands that computations be viewed as a static network of communicating tasks that operates as a data-driven systolic array. It is a good representative of many non-shared memory systems such as the Intel iPSC/2 or nCUBE 2. The major difference between Pringle and these other machines is that the latter two machines are based on the hypercube architecture which is a special case of the connections possible on the reconfigurable network of the Pringle. Also, the cost ratio of communication (cost of sending one floating point number) over computation (cost of one float point operations) is about 1 for Pringle and 200-300 for the two hypercubes.

By using many different heuristics, we derived the three programs shown in figure 3.1 that are believed to be optimal for the BBN butterfly, the Alliant FX/8, and the Pringle, respectively. The *block_transfer* and

† Developed originally by L. Synder and D. Gannon for an ONR sponsored project at Purdue University and Washington University.

inner_product used in figure 3.1 (a) are pseudo names of the built-in operations that BBN Butterfly provides. *Ch_x* used in figure 3.1 (b) is the channel variable generated by the compiler to pipeline the data transfer in Pringle. A more detailed explanation about these programs and how they are derived is given in chapter 8.

```
forall k in 1 .. P loop
        block_transfer(x, x_local, sizeof(x));
end forall;
forall i in 1 .. N] loop
        block_transfer(a[i, *], a_local, sizeof(a[i, *]));
        y[i] := inner-product(a_local[*], x_local[*]);
end forall;
```

(a) The BBN butterfly version.

```
forall k in 0 .. p-1 loop
        local tmp;
        for j in 1 .. m do
                tmp =  if (k==0) then x[j] else Ch_x[k-1];
                Ch_x[k] = tmp;
                for i in [k*n/p .. (k+1)*n/p] do
                        y[i] = y[i] + a[i,j] * tmp;
                end for;
        end for;
end forall;
```

(b) The Pringle version.

```
forall k in 0 .. n/32-1 loop
        local k1, k2 : int;
        k1 = k * 31 + 1;
        k2 = (k+1) * 32;
        for j in [1 .. m] do
                y[k1 .. k2] sum= a[k1 .. k2, j] * x[j];
        end for;
end forall;
```

(c) The Alliant FX/8 version.

Figure 3.1. *Three different versions of the matrix vector multiply program (for Butterfly, Pringle and Alliant, respectively).*

Each of these programs was derived from the original program by a sequence of correctness-preserving transformations. The state-space in terms of the program transformations for this program can be represented in a tree structure diagram which we called a decision tree for program transformations. The paths that lead from the original program to the above three programs are shown in figure 3.2. This diagram shows two interesting facts:

1. The complexity of the problem. The state space search diagram that we show is only a fraction of the whole tree that can be derived from the simple program.
2. The architectural dependence. The program transformation sequences for these three architectures are dramatically different. It is obvious that the differences in the sequences of the transformations are due to the differences in the architectures.

Some interesting questions arise from this example. What features of the architectures would cause such differences in selecting the program transformation sequences? How can these features be identified and integrated into the compilers? How does a compiler find these transformation sequences based on its understanding of the architectural features? What kinds of intelligence does the compiler need to make such decisions? How can we program the compiler so that it will have such intelligence? We will come back to this example to explain how architectural differences affect the program performance and how our system utilizes various heuristics to derive transformation sequences based on the features of the program and the architecture.

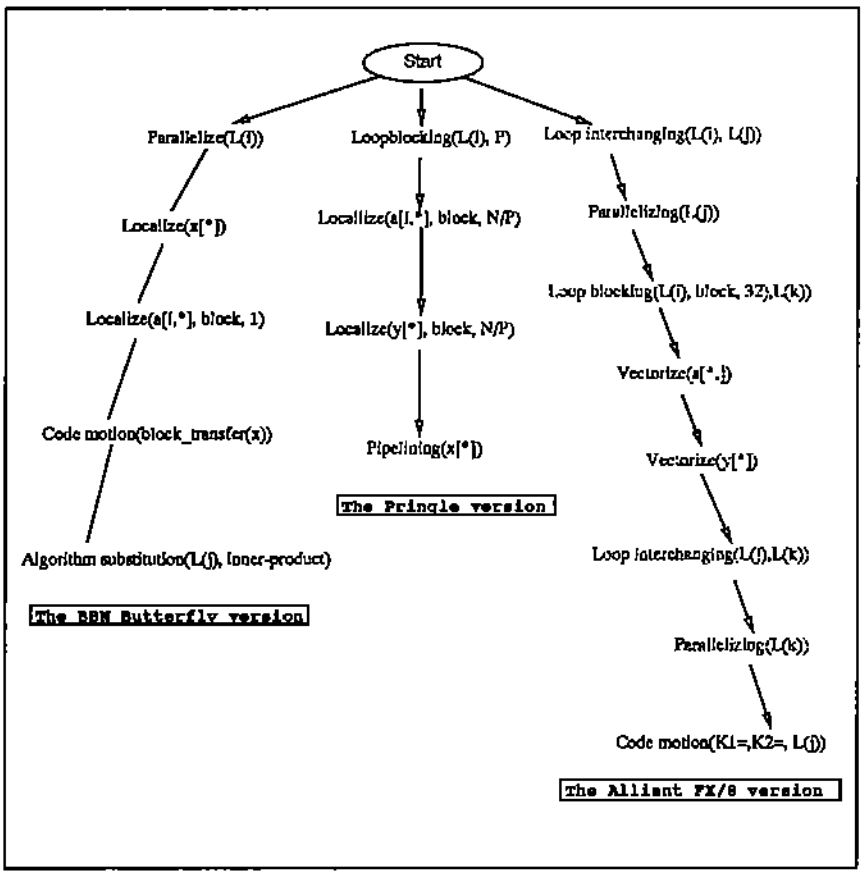

Figure 3.2. *The subtree of the decision tree for program transformation that shows the transformation sequence for the four architectures.*

If programs can be optimized for an architecture based on a particular program transformation sequence, this situation may not be too bad, since we can afford to spend a significant amount of resources, once and for all, to find this powerful sequence of transformations for this architecture. Unfortunately, the optimal sequence of program transformations depends not only on the target architecture but also on the program being optimized. Also, the behavior of the program not only depends on the architecture and the program, but also the input to the program. However, the static analysis of the compiler will not be able to do a good job on the input dependent behavior of the program. For this kind of program, the compiler should minimize the conditions that a computation depends on and generate some run-time tests to select the better approach.

The second example that we present is an attempt to parallelize a program to compute a fractal image called the Mandelbrot set on a 64 processor nCUBE 2. The Mandelbrot set arises from sequences of complex numbers defined inductively by the relation $z_{n+1} = z_n^2 + c$, where c is a complex constant. The behavior of the sequences depends on the parameter c and an initial value z_0. The Mandelbrot set is obtained by fixing z_0 to 0 and varying c. The nCUBE 2 is a distributed memory MIMD machine that utilizes the message-passing paradigm for inter-processor communication.

The Mandelbrot problem is a so-called "embarrassingly parallel" problem, because each pixel in the picture can be computed independently. The algorithm itself contains absolutely no interprocessor communication. Even though this is an ideal program to parallelize, we will demonstrate that an unintelligent parallelization can perform badly due to the poor load balancing.

The sample image that we used for the experiment is shown in figure 3.3. This image was chosen because the computation is not well distributed in the image.

Running the original program sequentially with the input we describe above on nCUBE 2 takes 1045.737 seconds (see column one in table 3.1). Since the program is a perfectly nested loop (looping through

the rows and pixels), it is obvious that the program can be optimized by parallelizing the outermost loop. A simple method for parallelizing the loops is to block the outermost loop to form P parallel tasks and distribute the tasks to processors. The speedup we get for using 32 processors is 16.601 (63.101 seconds) and the speedup for 64 processors is 32.881 (31.859 seconds). This result is certainly not satisfying. The major problem with this approach is the imbalance in computation loads, which is clearly shown in the variance of the elapsed time among all processors shown in column 2 and 5 in table 3.1.

A different approach that uses dynamic task scheduling to balance the load shows a very interesting result (figure 3.4). By distributing the tasks of 1024 pixels each, the resulting program is well balanced for different sizes of cubes (see the variance of elapse time in column 3 and 6 in table 3.1).

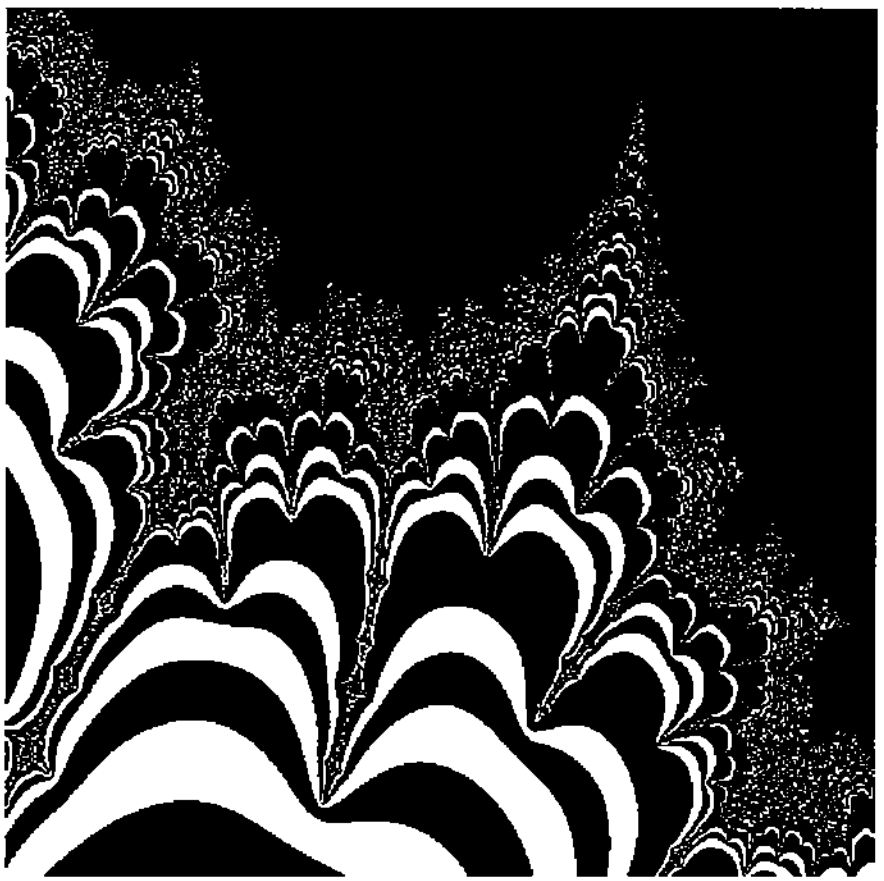

Figure 3.3. *The Mandelbrot set computed with* $p = [-1.781 .. -1{,}764]$, $q = [0.0 .. 0.013]$ *where the complex plane* $C = p + i * q$. *The black region in the upper-right corner represents the most computation-intensive area.*

Table 3.1. *Performance result of using cyclic, dynamic, and block allocations for the Mandelbrot set problem (number of processors = 32 and 64, task size = 1000 pixels).*

	Number of processors						
	1	*32(block)*	*32(dynamic)*	*32(cyclic)*	*64(block)*	*64(dynamic)*	*64(cyclic)*
Max. elapse	*1045.737*	*63.101*	*40.471*	*33.492*	*31.859*	*64.968*	*17.201*
Max. idle	*4.890*	*0.335*	*8.790*	*0.286*	*1.92184*	*1.066*	*47.558*
Vari. of elapse	*0.000*	*307.411*	*0.003*	*0.228*	*67.379*	*0.231*	*0.038*
Stand. deviation	*0.000*	*17.533*	*0.051*	*0.478*	*8.208*	*0.48*	*0.196*
Speedup	*1.000*	*16.601*	*25.839*	*31.277*	*32.881*	*16.096*	*60.901*

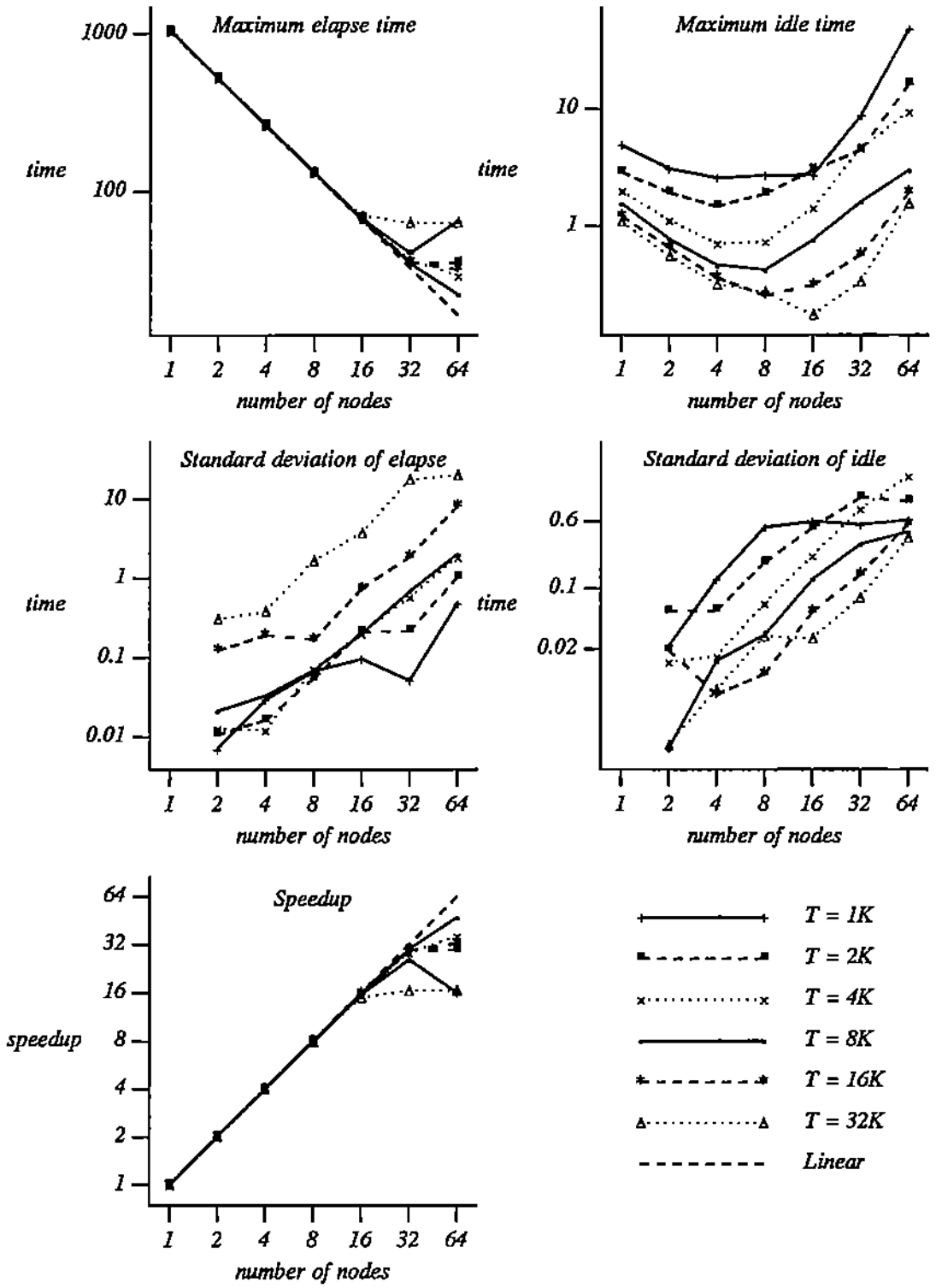

Figure 3.4. *Performance results of dynamic scheduling using different sizes of tasks and numbers of processors for the Mandelbrot set problem.*

As can be seen in the timing result in table 3.1, for the 32-processor case, the maximum elapsed time for dynamic task-scheduling is better than for the static blocking method. However, by doubling the processors to 64 processors, the result is actually slower than the 32 processor case. This slowdown is caused by the increase in the communication cost for larger cubes and the overhead of assigning tasks at run time.

A much better result is obtained for this particular example by blocking the outermost loop cyclically to form the tasks. The speedup for this method is 31.277 for the 32 processors and 60.901 for 64 processors (see columns 4 and 7 in table 3.1).

This example shows the characteristics of parallel programming: (1) Tedious details cannot be overlooked; (2) Gain in one area may cause loss in other areas; (3) Even when the sequence of transformations is known (in this case, to parallelize the outermost loop), the method for applying the transformations can still be very different. In this example, the compiler needs a fairly high level of intelligence and a good performance estimation tool to figure out the optimal task size (or cube size) to parallelize such an "embarrassingly parallel" problem. The final method is clearly better after the performance results are analyzed, but the challenge to parallel compilers is: how does the compiler figure this out without actually running the program?

The above two examples not only demonstrate the difficulties of parallel program optimization, but they also lead to the following observations:

1. Architectural characteristics have a significant impact on program optimization.
2. Different programs may require different sequences of program transformation to achieve optimal results on the same architecture.
3. Runtime behavior of programs affects the balance of the load, and the balance between the computation and the communication and cannot be ignored.

3.2. Methodologies for Improving Intelligence in Parallel Compilers

In this thesis, we coordinate three approaches to improve the intelligence of parallel compilers. These three approaches are a new paradigm for program optimization, systematic utilization of heuristics, and extensive application of AI techniques. We will discuss these three approaches in this chapter; the realization of the paradigm will be discussed in the next chapter.

3.2.1. Paradigms for Program Parallelism Improvement

Parallel compilers use program transformation techniques to restructure the control and data structures of the programs. Different sequences of program transformations lead to programs with different performance characteristics. One of the major tasks of parallel compilers is to choose on an appropriate sequence of program transformations so as to effectively map a program onto the target machine.

The key to achieving intelligent behavior in parallel compilers lies in appropriate paradigms for program optimization and sound methodologies for knowledge organization and integration.

Below we first examine existing models for selecting program transformation sequences and the problems with these approaches. We then introduce a new model called the feature-directed program optimization model.

3.2.1.1. Six Models for Selecting Program Transformation Sequences

Most existing parallel program optimizers adopt one or more of the following models:

1. *Compiler option model.* The compiler provides command-line options for users to choose a sequence of transformations among a set of pre-defined sequences or to specify a user-determined sequence of transformations. The Paraphrase system [KKLW80, KKLPW81] from the University of Illinois is an example of such an system.
2. *Annotation model.* The programmer directs the compiler to parallelize or vectorize certain program components (usually loops) or to decompose or distribute data in certain patterns by annotating the user program in forms of *user directives* or *assertions*. This model is supported by most parallel or vector compilers to make up for the shortcoming of the compilers.
3. *Predefined sequence model.* The compiler builder finds one or more predetermined sequences of transformations that are supposed to be optimal for the particular target machine and applies the fixed sequence on all programs. If there is more than one pre-defined sequence to choose from, the user may use either command-line options or assertions to select alternative sequence to use. Paraphrase[KKLW80] and many other parallel compilers support this model.
4. *Interactive model.* The compiler provides the programmer with an interactive programming environment and a set of program transformation techniques for the user to direct the program restructuring process

step by step and view the intermediate results of the transformations. Experienced parallel programmers may utilize this kind of compiler (usually referred to as programming environments) to produce highly optimized programs. PFC [AlKe84] from Rice university and Faust [GGJMG89] are two good examples of this model.

5. *Heuristics-driven model.* The compiler chooses the program transformations on the basis of heuristics. The quality of the knowledge base of the compiler determines the quality of the decisions that the compiler makes. Most parallel compilers utilize some heuristics, but very few rely exclusively on heuristics; none provide systematic processing of heuristics. In [Wang85], we presented a prototype expert system implementation of a parallel compiler that relies on heuristics and some user interaction to direct all its decisions.
6. *Program-directed model.* In this special case of the heuristics-driven model, the compiler selects transformations based on the patterns of the program dependence graph of the program. An example of this approach is presented in [WaGa89].

3.2.1.2. Analysis of the Models

One way to compare these models is to compare the effects of the paths that these models visit during the program optimization process.

The predefined sequence model will only visit a few predetermined paths out of the many possible paths in the decision tree. Due to the dynamism of the program behavior, the chance that the predefined sequence is the optimal path is rather small. The philosophy behind this model is that this model will achieve acceptable performance without expensive analysis for specific type of problems that the sequence is designed for.

The user annotation model and the interactive model rely on the user to select the transformations. Thus the part of the decision tree that is visited depends on the experience of the user and how hard he or she tries to optimize the program. The help the compiler provides is the mechanical program transformation and code generation.

The interactive model is superior to the user annotation model or predefined sequence model in the sense that programmers can base their decisions on the results of previous transformations and various programming tools can be incorporated into the environment to help users understand the consequences of their decisions. Some interactive parallel compilers also provide a limited degree of advice. However, the programmer is still the one who is supposed to make all the hard decisions. The interactive model gives the user better control of the optimization and parallelization but fails to remove the major burdens of parallel programming.

The heuristics-driven model is efficient in choosing the transformations. The number of transformation paths that the model examines depends on the size and quality of the heuristics. Some heuristics are rough and may cut off useful transformation paths accidentally. The quality of the compiler is determined by the richness of its knowledge base. Another problem of this model lies in the difficulty of collecting heuristics. Much effort has to be put into constructing a powerful knowledge base.

The program-directed model can be programmed to exploit the decision tree of program optimization systematically. It may achieve good results because the transformation sequence is chosen based on the structure of the program. However, the performance of a program depends heavily on the target architecture. Machine features need to be integrated into this model. Otherwise, the compiler will be machine dependent. The major problem with the program-directed model is that information that is not available at compile time may hinder the program optimization process. Heuristics for handling these situations or run-time tests need to be incorporated into the module to solve this problem. To summarize, none of the models alone have the capability of solving the complex program optimization problem. New models that support systematic analysis, incorporation of machine knowledge, knowledge organization and integration need to be designed so that intelligent parallel compilers can be built.

Why do most existing parallel compilers avoid systematic analysis and thus leave the hard decisions of parallelism improvement to the user? Among other considerations, we weigh the following as the two most significant factors in influencing the design decisions of compiler writers.

- Difficulty in obtaining expertise for parallel programming. Heuristics that application-programmers use are usually specific to the particular application and are not directly extendable to general problems that a compiler is facing.

- Too much computing power required for deciding on the optimal program-restructuring sequence at compile time. The computation and analysis of the program dependence graph also require a significant amount of computing power.

We will show below that with suitable infrastructure, the difficulties in obtaining optimizing expertise can be overcome and automatic knowledge acquisition is possible. With the rapid advances in workstation technologies and the call for utilization of parallel computing resources, using fast and cheap processing power of the workstations to optimize programs for supercomputers diminishes the problem of needing too much computing resources and makes this approach attractive. Also, by applying appropriate AI techniques, it is possible to cut down the decision tree to minimize the cost of optimization. The bottom line is: *the desire for improving parallelism automatically should not be put off by pragmatic problems that can be solved by suitable methodologies.* In the next section, we will present a new program optimization paradigm to solve these problems.

3.2.1.3. A New Paradigm for Improving Program Parallelism

As we discussed in the last section, a new program optimization paradigm is needed so that intelligent behavior can be observed in parallel compilers. In this section, we introduce a new paradigm for improving program parallelism that incorporates machine features into the decision-making process and provides a good foundation for the organization and integration of system knowledge.

A parallel architecture can be characterized by *machine features* which are properties of the machine that are related to the concurrent execution of user programs. Also, a program can be abstracted into a list of *program features* such as patterns or functionalities of the computation. By analysis of the heuristics for improving the program parallelism on the target machine, the heuristics can usually be attributed to certain features of the program and the architecture. On the basis of this understanding, we introduce a new program optimization paradigm that is called the *feature-directed program optimization* paradigm. Under this paradigm, heuristics are encoded with features of the target machine and the program. The control process that guides the program-restructuring process utilizes these heuristics to select the program transformations to apply. The program-restructuring process is an iterative process of selecting and applying the program transformation techniques to match a program to a particular parallel architecture. At each step of the process, the program is analyzed, a set of applicable transformations is chosen and compared based on some metrics, the most promising transformation is chosen and applied on the program, and the resulting program is evaluated. This process is repeated until the resulting program is "satisfactory." A performance evaluation unit can be defined from the match between the program and machine parallelism to serve as a metric for evaluating the transformations.

To realize this process, several questions have to be answered.

- *"What transformations to consider?"* Each transformation has different effects and purposes; it is not efficient to consider all possible transformations at each step. The selection of the set of transformations affects the efficiency of the optimization process. This selection should be based on the objective of the optimization. Heuristics to decide which transformations are more promising for certain tasks are needed to limit the search tree. To improve the efficiency, multiple transformations may be chained and treated as a new transformation in the iterative optimization process.
- *"How to select the most appropriate transformation?"* All transformations have a tradeoff and overhead. The decision for selecting a transformation will need to be based on the particular program, target machine and the current stage and objectives of the optimization. One possibility is to use the performance estimation to estimate the possible contributions the transformation can have on the concurrency. A heuristics-oriented rule-based system can also be used to make the decision. We will study the framework for control decisions in the next chapter.
- *"What are the effects of the target machine on the program transformation?"* The view of parallelism of the target architecture directly affects the execution of the program and must be considered when the transformations are chosen. To maximize the effectiveness of the compiler, the effect of the target machine on the selection of transformations needs to be studied carefully. This issue is studied in detail in chapter 5 and the paper [Wang88].
- *"When to stop the transformation process?"* For the same program, there are many different representations that have the same input-output semantics as the original program. It is impractical to try all of the sequences before choosing the best way to restructure the program. Heuristics, and some kind of metric,

must be employed in order to find the most promising transformation to apply at each step.

One major motivation for deriving the feature-directed program optimization paradigm is in its potential application in intelligent parallel compilers. This paradigm opens up many interesting research problems that have not been addressed before an intelligent parallel compiler can be implemented based on it:

- *Representation of the machine features.* How should the machine features be abstracted, represented and organized in the knowledge base and how are they integrated into the decision-making process?
- *Representation of the heuristics.* How do we acquire, represent, organize, and utilize the program-restructuring heuristics? What is the relationship between the program-restructuring heuristics and the machine features?
- *Choosing program transformations.* How does the system decide which program transformation to use at a particular stage of program optimization. What effects do architectural differences have on the selection of transformations?
- *Efficiency of the decision making process.* What techniques can the system utilizes to improve its efficiency and effectiveness? Can the compiler itself be parallelized and therefore able to utilize more computing power?
- *Learning.* Can the system learn from experience to improve its own ability? Can the framework provide hooks for a learning module?
- *Intelligent user interface.* When the system fails to come to a conclusion about a certain situation, can the system query the user in a intelligent manner? Can the system provide a friendly user interface?

A diagram illustration of this paradigm is shown in figure 3.5.

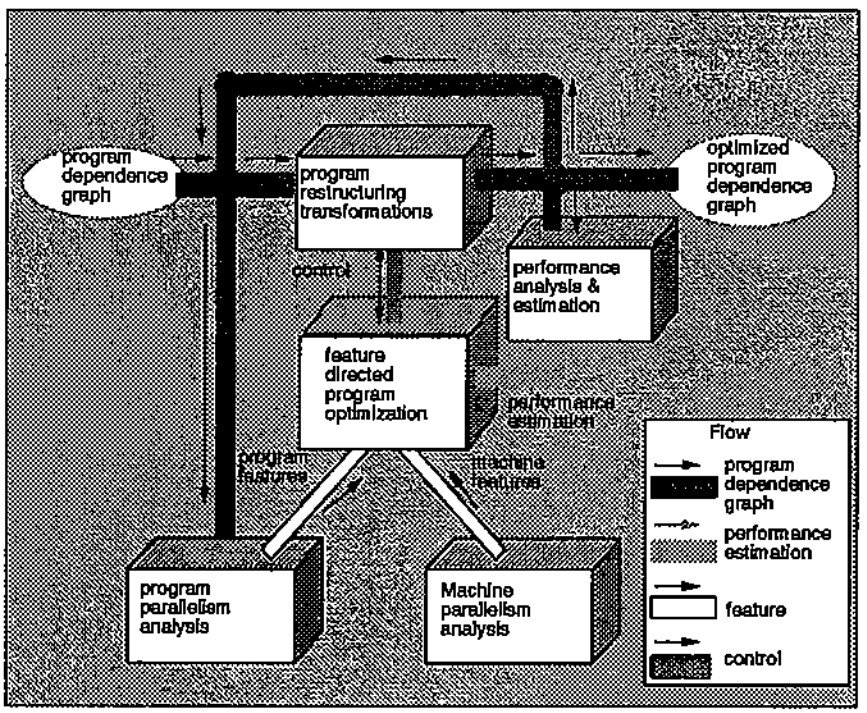

Figure 3.5. *The feature-directed program-restructuring paradigm. The black pipes represent the flow of the program dependence graph. The gray pipes represent the flow of the control information of the system. And the white pipes represent the machine and program features.*

Our approach to solving these problems and building intelligent parallel compilers can be outlined as follows:

1. *Feature-based target machine description.* The features of the parallel machines are analyzed and the target machines are described on the basis of their features. The target machine can be either represented in a heterogeneous or hierarchical structure. An object-oriented knowledge representation scheme is

described in chapter 5.

2. *Feature-based heuristics representation.* The program transformation heuristics are encoded on the basis of the features of the target machine and the programs. The heuristics are hooked to a hierarchy of machine features that is constructed by the system. This allows heuristics to be manipulated efficiently.
3. *Knowledge organization and integration.* The inference knowledge of the program transformation is organized into a structure that is called the heuristic hierarchy. This structure closely mimics the structure of the decomposed problem space. It also features competitive and opportunistic problem-solving methodologies. The heuristic hierarchy and methodologies of organizing and integrating this knowledge are discussed in section 4.3.
4. *Expert systems approach.* Since heuristics are used extensively to control the decision making, expert system technologies are used to build the intelligent program-restructuring system for program optimization and to use it as the central control unit of the intelligent parallel programming environment. Other expert systems, such as a machine knowledge manipulation expert system, an explanation expert system, and a program feature abstraction expert system can be incorporated.
5. *Intelligent program-restructuring based on feature-directed model.* The control process of program-restructuring is guided by features of both the target machine and the program. A knowledge base that contains a rich set of program-restructuring heuristics can be used to aid the control of the program-restructuring process. Users have the option to query the system about the decision-making process and if desired, to take over the control.
6. *Utilization of AI techniques.* AI techniques are used extensively to cut down the unnecessary branches in the decision tree and improve the efficiency of the system.
7. *Learning.* Learning models are being studied.
8. *Parallelism in the decision-making process.* Parallelizing the compiler itself allows the compiler to run on a more powerful machine and use more computing resources to solve the program optimization problem and to improve the quality of the generated code. In next chapter, we will discuss a program-restructuring framework which itself exhibits parallelism in the program-restructuring process.

3.2.1.4. Comparison to Other Parallel Program Optimization Models

Our paradigm for building intelligent parallel compilers and programming environments differs from traditional compiler approaches in the following respects:

- *Model of parallelism optimization.* Most compilers and parallel programming environments either use predefined program transformation sequences or rely on users to make the major program-optimization decisions. In our system, we utilize the feature-directed program-optimization model which opens up a completely new avenue for research into intelligent parallel compilers.
- *Organization and integration of the knowledge.* The knowledge is explicitly represented in our system but implicitly hidden in most conventional parallel compilers. In our system, knowledge about the target machine is encoded and employed in terms of machine features. Specification, organization, integration and utilization of the heuristics are all based explicitly on the machine features and program features.
- *Systematic analysis.* The feature-directed program optimization model uses systematic analysis and automatic program transformation, while most other parallel compilers rely on ad hoc heuristics.
- *Degree of the user involvement.* Our reasoning model provides a mechanism for the user to intervene in the decision-making process. This allows the system to adjust to suit different experiences and capabilities of different users.
- *Extensive utilization of AI techniques.* Our approach makes extensive use of AI techniques and knowledge manipulation methodologies for parallel compilers. heuristics.
- *Multiple target machines and knowledge generalization.* One key ability of an intelligent parallel compiler lies in the ability of transporting and integrating experiences learned from a particular machine to others. The integration of knowledge for optimizing different kinds of parallel architectures into one system and the use of machine features as a basis for knowledge representation ease the problem of knowledge transfer and knowledge generalization. Accumulating the abilities of the system can greatly enhance the capability of the system as the development of the system progresses. This feature is particularly valuable for parallel compilers since the cost of building such a system from scratch for each

individual target machine is so high. With this approach, only the back-end (code generation) needs to be specialized for the target architecture. More important, users' parallel programs can be immunized from machine-dependent constructs to preserve portability without sacrificing efficiency.

To summarize, the combination of expert systems, knowledge acquisition, and AI techniques for analysis, collection and accumulation of the system knowledge provides a practical alternative to traditional parallel compiler approaches. This paradigm can be used to build compilers that are far more powerful than even the best parallel programming environments available today. On the other hand, with this approach the need for efficient decision-making processes and new methodologies for representing, organizing, integrating and utilizing the knowledge becomes even more important. Methodologies for applying state-of-the-art AI techniques to these problems to realize this new framework in the construction of parallel compilers is studied in [WaGa89], and in chapters 4 and 7 of this thesis.

3.3. Utilization of Heuristics

Heuristics are methods, criteria or guidelines for selecting promising actions among alternatives. They are usually referred to as simple principles that we learn from experiences or experiments†.

Heuristics may not always lead to effective solutions, but they represent the compromises between simple fast guesses and actual but expensive evaluations. Heuristics are usually effective in the following situations: state-of-the-art decision-making in non-well-defined problem domains, short-cut principles for expensive operations, and approximated substitutions for non-attractive exponential time-bound algorithms.

The state space of the problem consists of facts that are derived from the problem domain or intermediate conclusions of the rules used in solving the problem. A heuristic can be translated into a function which maps a state of the decision-tree into another state by adding new facts or modifying existing facts. In expert systems, a heuristic may be represented in one or more rules in the knowledge base.

3.3.1. Systematic Discovery of Heuristics

Heuristics may be discovered systematically by consulting simplified auxiliary models of the original problem ([Pear84]). By simplifying the problem, the hope of finding a solution is higher and the cost should be lower. Also, since the problem being solved is identical to the original problem except for a few constraints, it is generally true that the solutions to the simplified model are likely to be admissible to the original problem.

Although systematically relaxing constraints of the problem can lead to the discovery of good heuristics in many cases, relaxing a few constraints in a complex problem domain may not be enough for simplifying the problem. In general, the relaxing process can be repeatedly applied to the relaxed model until a solution is easy to find. However, relaxing more constraints means that the differences between the relaxed problem and the original problem are more significant; the heuristic derived from the relaxed problem may no longer be effective in the original problem. Therefore, any new heuristics self-learned by heuristic relaxation should be subject to the inspection of the knowledge engineers.

3.3.2. Translating Heuristics into Rules

One major source of obtaining heuristics is through case studies; most program transformation heuristics are derived through careful study of the problems at hand. Problems of knowledge acquisition in case studies include recognizing the factors involved in the decision-making process and the similarities between the particular case and more general cases. In figure 3.6, we describe a process of abstracting domain-dependent knowledge.

For each heuristic, there are actually many ways to represent it in the knowledge base. The proper choice of encoding is critical for the knowledge to be effective. To illustrate the complexity in transforming heuristics into rules, we show an example for evaluating the match of a program fragment to the architecture.

Heuristic 3.1. When mapping a loop into parallel tasks, it would be better if the size of the loop, N, is a multiple of the number of processes, P, if the loop size is small. This factor is less important when the loop size, N,

† In this thesis, the term "heuristics" is generalized into any simple, effective or efficient methodologies used in decision-making. Therefore, a heuristic may be an expertise that comes out of the state-of-the-art experience of an expert or a well-defined algorithm.

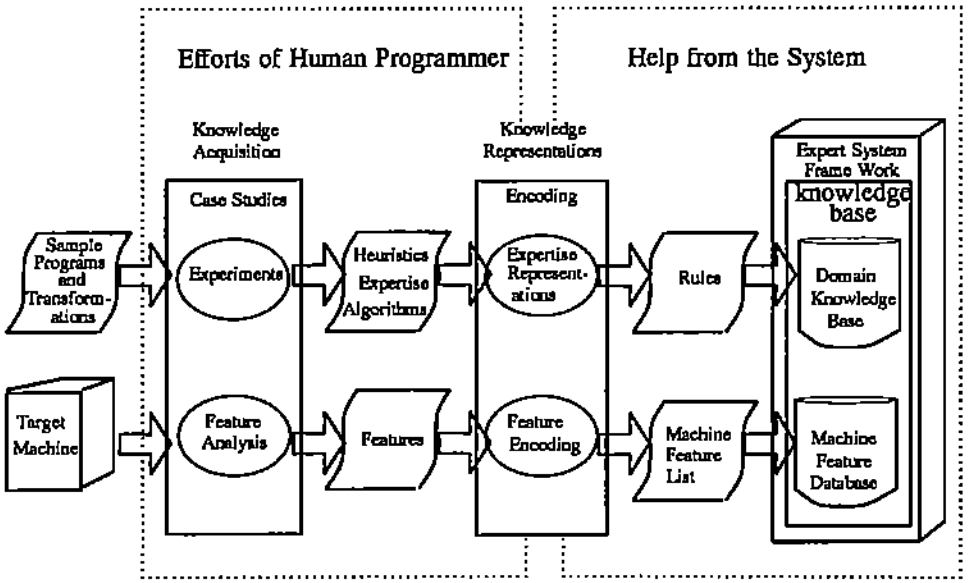

Figure 3.6. *Process of abstracting a domain dependent knowledge base.*

is large compared to P.

To translate this heuristic into rules, the following formula is used to capture both conditions in Heuristic 3.1.

$$V = eval(N, P) = \frac{N/P}{\lceil N/P \rceil}$$

This formula quantifies the heuristic quite nicely. N/P and $\lceil N/P \rceil$ are both integers differing by at most one. The result of the real division is 1 when N divides P or approaches 1 when N / P is a large integer.

On the other hand, this encoding represents a non-trivial heuristic in knowledge acquisition. It is very difficult for the system to perform this encoding without help from human experts. A good knowledge acquisition system may ease this difficulty but may not solve the problem completely. This is why human assistance is needed in the above knowledge abstraction process.

3.4. Applying AI Technologies to Parallel Compilers

From our experience, there are many areas where AI techniques may help in the construction of parallel compilers. A list of the AI techniques that we have employed in the implementation of our prototype system is listed below: We briefly discuss some of them here.

- Search algorithms. In chapter 4, we will discuss the use of some AI search algorithms, such as A^*, for searching through the decision tree in the program optimization process. These AI search algorithms use heuristics to cut down the unnecessary traversal of states in the decision tree to improve efficiency. With suitable performance estimation functions, the A^* algorithm can find optimum solutions.
- Goal reduction. Goal reduction techniques (such as forward-chaining, backward chaining, hybrid methods, etc), can be used as goal searching and processing and to cut down the search trees to improve efficiency in decision making, Furthermore, resolution and unification can be used to deduce the search goals. The theorem-proving system can be used to deduce and analyze heuristics and find the inconsistency in the knowledge. The *hier-blackboard* model discussed in section 4.4 is a framework for goal reduction.
- Constraint propagation and satisfaction. Static analysis of the program has its limits. For instance, a program dependence test may be obscured by a variable in a loop bounds, or the task decomposition may be crippled by the unknown loop bound in the outermost loop. Some of these decisions can be postponed until run time. To ensure that only the minimal run time test is generated, constraint propagation can be used to propagate the critical conditions for such run time tests at the needed points.

- Planning. Using planning for selecting a program transformation sequence or generating the schedule of parallel execution is discussed in section 4.2.
- Generate and test. The model consists of a generator and a tester, where the generator generates a number of possible cases, the tester eliminates the inapplicable or non-promising ones. An example is to apply this technique to data decomposition, where a data decomposition generator can generate possible decompositions of arrays and an examining expert can be used to eliminate less promising compositions for limiting the selections. In [Wolfe82], generate-and-test technique is also used in deciding the plausible loop order for interchanging perfectly nested loops.
- Pattern recognition. Pattern recognition can be used to recognize the program parallelism, machine features, and opportunities for improving parallelism. An example of using pattern recognition to abstract program features involves recognizing opportunities for pre-optimized algorithm substitution (see chapter 7). A sample program to recognize a generic pattern of inner product operations was given in chapter 2.
- Man-machine interface. Advances in the man-machine interface such as natural language processing and visual programming can be used to achieve intelligent user-interaction.
- Knowledge engineering. Knowledge representation and manipulation techniques can be employed to represent, organize, and integrate the program transformation heuristics and manipulate machine features. Object oriented knowledge representation of machine features and the organization of heuristics are discussed in chapter 5.
- Problem-solving models. Problem-solving models such as rule-based systems, blackboard systems, and object-oriented models can serve as frameworks for reasoning in intelligent compilers. This problem is discussed in detail in chapter 4.

3.5. Conclusion

In this section we discuss some challenges that a parallel compiler faces. A new paradigm for program optimization is presented. Heuristics and the application of AI techniques in parallel compilers are also briefly discussed.

CHAPTER 4
A FRAMEWORK FOR THE CONTROL OF INTELLIGENT PARALLEL COMPILERS

In chapter 3 we proposed a new paradigm for parallel program optimization. In this and the following three chapters, we will discuss methodologies for realizing the feature-directed program optimization paradigm into intelligent parallel compilers. We first decompose the problem of program optimization into hierarchical structure of subproblems. In this chapter, the framework for implementing the paradigm is discussed. The problem is first formulated into a planning problem. Then three different frameworks for implementing the paradigm are presented and their advantages and disadvantages are discussed.

4.1. Parallel Program Optimization as a Planning Problem

The program optimization system can be viewed as a planning system. A planning system is a program that develops a course of actions, or a *plan,* to reach a desired goal. This plan is then used to guide the execution of planned activities. When the activities represented in a plan are timed, the plan is called a *schedule.*

Formally, a planning system *PS* can be defined as a quadruple:

$$PS = (S, OP, S_0, S_G)$$

Where S is the set of problem states, OP is the set of operators defined by a state-transition mapping from one state to another, S_0 is the initial state and S_G is a set of goal states. The planning process is to find a plan Ψ (which is a sequence of operators) that will transfer the initial state to one of the goal states. That is:

$$S_0 \overset{\Psi}{\rightarrow} S_g, \quad \text{where } S_g \in S_G.$$

Ψ is actually a sequence of operators that change the problem states:

$$S_0 \overset{\Psi_1}{\rightarrow} S_1 \overset{\Psi_2}{\rightarrow} S_2 \overset{\Psi_3}{\rightarrow} S_3 \ \ldots \ \overset{\Psi_n}{\rightarrow} S_g$$

Where Ψ_i's are operators that map S_{i-1} into S_i, that is, $\Psi_i(S_{i-1}) = S_i$. we have $\Psi(S_0) = S_g = \Psi_n(\Psi_{n-1}(\Psi_{n-2}(\cdots \ \Psi_1(S_0) \ \cdots)))$, or:

$$\Psi = \Psi_n \bullet \Psi_{n-1} \bullet \Psi_{n-2} \bullet \cdots \Psi_1.$$

A parallel program (represented as an augmented program dependence graph) is a rough schedule for concurrently executing the computation specified in the program on a parallel architecture. The program dependence graph represents the constraints that the operations on the program must respect. The schedule generated by the compiler statically determines the flow of the control, the flow of the data, and the utilization of resources. The parallelism optimization process modifies the *schedule* of the operations to improve the performance of the program on the target machine.

There are many ways to formulate a parallel compiler as a planning system. The planner can use the dependence graph as constraints for generating plausible execution plans. Alternatively, programs can be viewed as problem states and transformations as operators to refine the states. For each node in the decision tree of the program optimization process, the children of the nodes are applicable transformations that can be applied to the program. The planner's objective is thus to generate a plan for refining the execution plan, or more specifically, to generate a list of transformations to improve the parallelism of the program. The program optimization process can be represented by the planning problem $PS = (S, OP, S_0, S_G)$, where S is the set of program dependence graphs, OP is the set of program transformation techniques, S_0 is the original program,

and S_G is the set of optimized programs. The objective of the program optimization process is to find an appropriate sequence of transformations $\Psi_n \bullet \Psi_{n-1} \bullet \Psi_{n-2} \bullet \cdots \Psi_1 = \Psi$ to translate the program dependence graph into an optimal form for the target machine.

Once the program optimization problem is defined from the state-space transformation paradigm, the problem is then to find a solution path in a search tree whose nodes are programs and whose arcs are program transformations that modify the programs. The key issue is the selection of the most appropriate operator to apply at the given state, represented as a node in the search tree. We shall now examine how a plan Ψ, the sequence of transformations, can be obtained. This approach differs from an ordinary planning problem in two respects: first, there is no clear definition of the goal states; second, even though optimization is desired, the user may not be able to afford the cost of finding the optimal solution, since the cost of verifying the applicability of the transformation is usually fairly high. In this case, a partial solution may be acceptable.

To guide the selection of the operators and identify the goal states, a performance objective function, E, is defined. The function E is a mapping from the state space to a real number which represents the performance measure of the states. This performance objective function is usually based on certain heuristics. Search algorithms based on the performance evaluation function are called heuristic-guided graph-search algorithms. The objective of the problem optimization process is then to find the transformation sequence Ψ, $\Psi = \Psi_n \bullet \Psi_{n-1} \bullet \Psi_{n-2} \bullet \cdots \Psi_1$, which transfers S_0 into S_g such that $E(S_i) \geq E(S_g)$ for all $1 \leq i < n$. The definition of this performance objective function is subject to the degree of optimization, the affordable resources, the knowledge of the architecture, etc. Modifying this function will change the characteristics of the program optimization system.

4.2. Frameworks for Realizing the Feature-Directed Program Optimization Paradigm

Below we will examine three different frameworks for realizing the feature-directed program optimization paradigm. The first approach is to use heuristic-guided state space search algorithms. Under this approach, the program optimization process is to find the optimal path in the decision tree of the program transformation process. The algorithm searches through the decision tree to find the goal states and uses some heuristic functions to prune subtrees that are not promising. A good representative of such algorithms is the A^* algorithm. As will be explained below, the A^* algorithm is a variant of the *best first search* algorithm that uses a heuristic function to estimate the cost of going from the starting state to a goal state in the decision tree. A^* is examined here because it guarantees that the algorithm will finish with an optimal solution.

The second approach, the *heuristic hierarchy* approach, and the third approach, the *hier-blackboard* approach, both utilize the hierarchical knowledge organization but with different flavors. The hierarchical knowledge organization is preferred over the flat structure of rule-based systems because heuristics are often fragmented during the knowledge acquisition process. Also, this loss of information (about the relationship between the rules) cannot be recovered after the knowledge is translated into rules. The hierarchical knowledge organization retains the relationship between rules in the knowledge base. The *heuristic hierarchy* utilizes the hierarchical knowledge organization but also provides a control and reasoning strategy. The *hier-blackboard* model is an extension of the *heuristic hierarchy* with the additional power of opportunistic reasoning and parallel processing. We will discuss these three frameworks and compare them in detail below.

4.2.1. Heuristic Guided State-Space Search

The selection of transformations may be guided by heuristic functions or other systematic approaches such as rules in a rule-based system. or instance, algorithm A^* [Nilsson80] can be incorporated. The algorithm A^* is a variant of the *best-first* search of a problem graph. It transforms the planning process into a graph search problem guided by a heuristic function f. The evaluation function $f(S_i)$ at any node S_i estimates the cost of the minimal path from the start node S_0 to the node S_i (denoted $g(S_i)$) plus the cost of a minimal cost path from node S_i to a goal node S_G (denoted $h(S_i)$). That is, $f(S_i)$ is an estimation of the cost of a minimal cost path constrained to go through node S_i.

$$f(S_i) = g(S_i) + h(S_i). \tag{4.1}$$

At each stage of the node expansion, the algorithm chooses the node that achieves the minimal evaluation function to expand. This algorithm is called Algorithm A.

Let $cost_{\min}(S_i, S_j)$ be the actual cost of a minimal path between the two nodes S_i and S_j. The function $h^*(S_i)$ is defined to be the cost of the minimum cost path from node S_i to any of the goal states.

$$h^*(S_i) = \min_{S_g \in S_G} \left\{ cost_{\min}(S_i, S_g) \right\} \tag{4.2}$$

Also define $g^*(S_i) = cost_{\min}(S_0, S_i)$, which is the cost from a start node to the node S_i. And $f^*(S_i) = g^*(S_i)+h^*(S_i)$ is the cost of an optimal path from S_0 constrained to go through node S_i. When the estimation function h is a lower bound of h^*, this algorithm is called A^* [Nilsson80].

To map the algorithm to the program optimization process, the function $g(S_i, M)$ is the estimated performance of the program S_i on the target machine M. And $g^*(S_i, M)$ is the actual performance of the program S_i on the target machine M. The function $h(S_i, M)$ is the estimated performance improvement the program optimization process can have on the program S_i on the machine M. And the function $h^*(S_i, M)$ is the maximum improvement that program optimization can achieve on the target machine M. This definition realizes the feature-directed program optimization paradigm because the features of the program and the machine are casted into the performance estimation of the program on the machine.

Lemma 4.1. Algorithm A^* will terminate if there is a path from S_0 to a goal state S_g [Nilsson80].

Lemma 4.2. Algorithm A^* is admissible (that is, A^* will terminate by finding an optimal solution if there is a path from S_0 to a goal state S_g) [Nilsson80].

The efficiency of the A^* algorithm depends on the choice of the evaluation functions. The precision of h depends on the amount of heuristics it possesses. When $h = 0$, it reflects complete absence of any heuristic information about the problem and results in a breadth-first search. However, since such an estimate is a lower bound on h^* the algorithm is still an admissible algorithm. Another interesting property about A^* is that the more "informed" the algorithm is, the fewer nodes it will expand. This property is described in the following Lemma.

Lemma 4.3. Let A_1 and A_2 be two versions of algorithm A^* that use different evaluation functions. If A_2 is more informed than A_1 (i.e. $h_1(S_i) \le h_2(S_i)$ for all i), then at the termination of their searches on any graph having a path from S_0 to a goal state S_g, every node expanded by A_2 is also expanded by A_1. It follows that A_1 expands at least as many nodes as does A_2 [Nilsson80].

The major question in applying the A^* algorithm involves defining the evaluation function and goal states. There are many ways to define the evaluation functions; the more accurate the estimation is, the fewer nodes the algorithm has to visit. On the other hand, accurate estimation of the program performance is usually very expensive to obtain. So the proper choice of the estimation lies in the compromise between the cost of computing the evaluation function and the cost of traversing the node (applying the transformations). More details about evaluation functions and their application in controlling the program optimization is discussed in detail in chapter 6.

Here we give a simple example. One heuristic for estimating the execution time involves using the operation counts. For a sequential statement block, this number is the sum of the number of the operations in each of the components. For loops, this is the number of iterations times the number of operations inside the loops. For conditional statements, a probability value can be assigned to each branch and the operation count for each branch is the number of operations in the branch times the probability that the branch would be taken. The operation count for the condition statement is then the maximum number of operations of the two branches. The operation count for a parallel loop is defined to be the maximum number of operations in each of the parallel tasks. This estimation function described above is rough but cheap to compute. Note that architecture dependencies such as the synchronization cost, memory utilization, and cache miss-ratio are all ignored in the estimation. However, this can serve as a framework for more sophisticated performance estimation. For example, for memory optimization, the operation count for a statement can be replaced by the cost of memory accesses in the statement. The sequential operation count OC^1 is the total number of operations to be performed in the program and the parallel operation count OC^P is the number of operations for concurrent execution on P processors. The speedup based on this performance estimation function is thus defined to be $SP_{OC} = \frac{OC^1}{OC^P}$. For algorithm A^*, the cost of the arc between a pair of nodes is defined to be the changes in operation count that the transformation would have on the program. And the heuristic functions g, and h are defined as $g(S_i) = 0$ and $h(S_i) = SP_{OC}$.

Lemma 4.4. Algorithm A^* defined above using the estimation functions g, h and f is admissible.

Proof: Since the arcs in the state space graph represent changes in operation counts, the cost of the path from S_0 to node S_i is the operation count of the program S_i. Subsequently, the function $h^*(S_i)$ is identical to the function $h(S_i)$. And the function h is always a lower bound on h^*, so by lemma 4.3 we know that the A^* algorithm will always find a program that would generate most parallel statements.

QED.

Since this heuristic ignores the impact of synchronization cost and differences in cost of different operations, the "optimal" program generated by the algorithm may not be optimal for general programs. However, this heuristic is useful for "embarrassingly parallel programs" since the lack of communication between the processes makes factors ignored by this heuristic unimportant, and the parallel operation count is thus a good indication of the concurrent performance.

Another possible way of defining the heuristic function is to define the function g to be the estimated performance of the program based on the current state, and the function h to be the estimated potential improvement for this state.

4.2.1.1. Non-linear Planning and the Coordination of Multiple Thread State-space Search

For a finer formulation of the planning problem where each node represents the scheduling of an operation, the goal of the system is to generate an execution plan. This problem can be translated into a multiple-task scheduling process where nodes in the dependence graph are the operations to be performed and dependence arcs define the precedence relations between the operations. The output of the planning system is a P-thread parallel program with annotations for data decomposition and allocation. The control threads of the program compete for shared resources such as the memory, communication network and processors but cooperate to carry out the objectives of the program through synchronized communication. The resulting plan is partially ordered since the dependence relations need to be respected. This type of planning is commonly referred to as *non-linear planning* [Sacer77, Tate76, Wilkins84]. The complexity of the search algorithm depends on the number of nodes generated. The size of the search tree is bounded by b^d, where b is the *branching factor* and d is the depth of the tree. Obviously, the search tree for parallel program optimization grows very rapidly. To overcome this complexity, the problem can be decomposed into P subproblems where each subproblem plans the execution for a particular processor. The algorithm A^* can be applied to generate the plans for the subproblems. The coordination between these P planning processes is a global strategy to avoid violating precedence constraints and resource-conflicts. The interaction between the processors is the data dependence between the blocks of statements assigned to each of the processors. The cost of an arc is defined to be the time to perform the operation that the arc points to. Communication and synchronization time needs to be added to the processing time of the operation if data must be obtained from remote processors. For this problem, the performance evaluation function f is defined to be $g + h$, where $g(S_i)$ is defined to be the time it takes to perform operations from the beginning up to operation S_i, and $h(S_i)$ is the estimated execution time for the remaining operations. The parallel execution time is the maximum execution time of each of the processors. The goal of the compiler is then to find a schedule (a parallel program) so that the parallel execution time of the program on a P processor machine is minimum.

Depending on the granularity of the schedule, the program dependence graph can be abstracted at different levels. It is generally impractical to schedule programs at a fine grain level, since this multiple task-scheduling problem is an NP-complete problem [Sarkar87] and there is a big overhead for scheduling large programs. Even at the task level, the problem is still relatively complex.

To summarize, the state-space search approach for parallel program optimization applies a heuristic function as the guideline for searching for the solutions. Heuristics are quantified into heuristic functions to help to choose the "best" node to expand. As indicated by Lemma 4.3, the quality of the algorithm depends on the quality of the heuristics used and the quality of the quantification of the knowledge. The advantage of this approach is clear: systematic processing is possible. Also, the behavior of the algorithm can be controlled by the heuristic function so that the characteristic of the algorithm can be modified by changing the heuristic functions based on the objective of the system. Based on this model, new heuristics can be tested or compared to existing heuristics. On the other hand, the disadvantage of the approach is that it is not always possible to characterize the heuristic by numerical values. Distortions to the heuristics are likely in the process. Also, for complex problems that involve many different heuristics at different stages of the problem (such as the problem for program optimization), trying to merge all heuristics into one heuristic function and applying it to every step in the search process is difficult and inefficient.

One important characteristic of the above approach lies in the specification of the problem. The separation of control knowledge (planning), data (state descriptions and goals), and problem-solving knowledge (operators) allows the system to be adaptive to different objectives of the problem at different stages of problem solving. This suggests an alternative approach which hierarchically decomposes the problem into subproblems and uses different sets of rules that are specialized for the subproblems in order to select the best node for the particular stage of the problem-solving process to expand. We will first look at the structural organization of the problem domain and then discuss two frameworks that base on this approach.

4.2.2. Hierarchical Decomposition of the Parallelism Improving Problem

The problem of program optimization is a very complex problem, therefore, decomposing it into simpler subproblems and coordinating the subproblems to solve it is beneficial. In this section we present one such decomposition method and this decomposition will serve as the foundation for the discussion of two other frameworks *heuristic hierarchy hier-blackboard* which are discussed in sections 4.2.3 and 4.3.5, respectively.

The process of program parallelism optimization can be classified hierarchically into three problem solving modules that we call the *parallelism-defining* layer, the *parallelism-matching* layer, and the *parallelism-matching control* layer. Depending on the complexity of the subproblem, each of these three layers may be further decomposed into finer subproblems.

The parallelism-defining layer abstracts the program parallelism and the machine parallelism into lists of machine and program features. The parallelism-matching layer matches the program onto the target machine by performing a sequence of program transformations. Since a single transformation may serve different purposes, it may belong to different categories. Therefore, we separate the heuristics in the program restructuring control layer into two sub-layers: the *program restructuring subgoal selection* layer and the *transformation* layer. The transformation layer contains the transformation techniques which we term *transformation modules.*

Each transformation module consists not only of the description of the transformation technique, the conditions for the transformation to be applicable and the procedures to carry out the transformation, but also the heuristics pertaining to the feasibility of the transformation under various circumstances, short-cut rules for applying the transformation, methods for estimating the effects of the transformation, etc.

The program parallelism improving process can be decomposed into the following five subproblems.

- *Improving general program parallelism.* The major purpose of this process is to improve the structure of the problem to prepare for other processes below. This goal can be achieved by reducing the amount of data or control dependence present in the program dependence graph. Machine-independent transformations for removing redundant code, breaking dependence cycles, and improving localities can be applied.
- *Creating tasks.* The aim here is to decompose the control structure of the program so as to create tasks and vector operations. One major consideration involves balancing the load of the created tasks.
- *Scheduling tasks.* The scheduling of the tasks/processes is another important factor in obtaining optimal performance. Traditionally, this problem is viewed as the task of the operating system. However, studies have shown that static estimates done at compile time can simplify the task of the operating system at run time [Cytron84]. There are techniques (e.g. do-across) that can estimate the required minimum process delay time to reduce unnecessary memory traffic due to pulling the lock variables prematurely. Compile time analysis can also help to decide what run-times to generate.
- *Minimizing synchronization.* When a sequential program is mapped to a multiprocessor machine, the proper synchronization operations must be inserted in the code in order to preserve the semantics of the original program. Synchronization costs penalize the program performance, and, in the worst case, may serialize the whole computation. Fewer synchronization points mean less processor idling time and better overall system performance. Grouping closely related micro-tasks into one task, copying repeatedly used data into local memories, and changing data access patterns may have a positive effect on minimizing the synchronization cost.
- *Optimization of memory accesses.* Since the data access time for different components of the memory hierarchy may be different, the utilization of fast memory components (like cache) and the removal of unnecessary data accesses will shorten the access time and speed up the computation. Array decomposition, data copying, scalar gathering, strip-mining, loop interchanging, loop blocking, and other transformations can be used to achieve a lower cache miss ratio and improve locality.

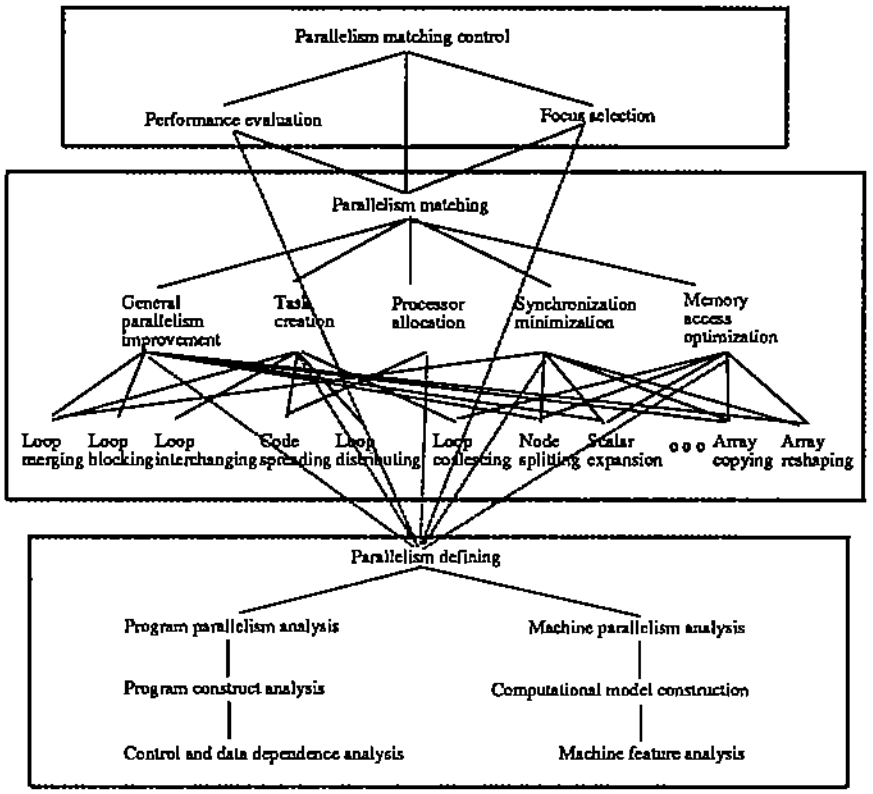

Figure 4.1. *A hierarchical decomposition of the process of optimizing program parallelism.*

Each of these five subproblems may select any of the transformations in the underlying transformation layer. The selection of the transformations is based on the heuristics in the transformation layer and the features defined in the parallelism-defining layer. Since these five problems are interrelated, the restructuring control process coordinates the interaction between them.

The program fragment under consideration is called the *current focus* of the system. A program is decomposed into a list of focuses by the *focus selection process. The topmost layer of the hierarchy is the parallelism-matching control layer. It selects the current focus and controls the optimization of the current focus.*

Global coordination between different focuses is often needed. For example, the memory access optimization subgoal will try to optimize the memory accesses and decompose the array storages based on the program focus and the machine model to which it is assigned. The array decompositions chosen in the subgoal may be changed when global consideration and adjustments are made.

The performance evaluation process evaluates the performance of transformations on the program focus and provides quantified evaluation for the parallelism matching control layer to make decisions.

This hierarchical decomposition of the program parallelism optimization problem divides the problem into interacting processes based on the hierarchical structure we described above. It models the conceptual interactions between different functions of the program restructuring process into a concrete structure so that controls in these functional units can cooperate and interact with each other. This hierarchy decomposition also allows specialized heuristics to solve the problem so it can significantly improve the flexibility and efficiency of the transformation process. It also provides a model for the decomposition and organization of the heuristics.

4.2.3. Heuristic-Guided Reasoning and the Expert Systems Approach

For the rule-based approach, at each stage of the program optimization process, the control of the process utilizes a set of *rules* to decide how to restructure the program. A prototype implementation that used the rule-based expert system approach was reported in [Wang85]. In this experiment, control heuristics were

encoded into Prolog predicates which chose the "most appropriate" program transformation to apply. This experiment with an expert system achieved mixed results. While the system was able to generate efficient code for some particular programs, it failed in many other cases. The major drawback of the system was its lack of heuristics. It employed only about three dozen rules for choosing the transformations. On the other hand, this experiment exposed a problem common to flat-structured first generation rule-based systems: the fragmentation of the knowledge and lack of systematic knowledge acquisition tools. This makes enhancing the ability of the system a very involved process. We concluded that structured organization of the knowledge is necessary and that systematic integration of the heuristics is a key issue to knowledge enhancement and automatic learning of the system. The *heuristic hierarchy* reported in [WaGa89] was our first attempt in moving to more intelligent expert systems.

4.2.3.1. The Heuristic Hierarchy

While the modularity and integratability of the rule-based expert systems make modifying the knowledge base easy, its opacity of knowledge and inefficiency in execution are the major drawbacks. For example, translating a heuristic into a set of rules causes the knowledge to be fragmented across the rules. Even though there may be strong relations among many of the rules, the fragmentation causes an unfortunate loss of coherence. Furthermore, this makes maintenance and modification of the knowledge base difficult.

To improve the integration and modularity of the knowledge, we organize the heuristics based on the decomposition of the problem-solving methods. This organizes the rules into a hierarchical knowledge structure called the *heuristic hierarchy* [WaGa89]. A heuristic hierarchy consists of one or more hierarchical *layers*. Based on the functionalities of the rules, rules in the same layer are divided into groups of rules that are called *actions*. Each action has a goal associated with it; invoking the action is an attempt to accomplish the goal of the action. The top layer of a hierarchy contains only one action, which is the entrance point of the control flow, and the goal of this action is the goal of the hierarchy. The *heuristic hierarchy* is a way to simplify the modeling of the problem into structured units. Layers in the hierarchy represent the conceptual hierarchical levels of the problem-solving process where in each layer the different actions represent possible solution steps that can be utilized to achieve the goals of the subproblem that the layer faces. The *heuristic hierarchy* integrates rules into conceptually and logically related units whose relationship reflects the control flow of the problem solution. Horizontal relations among the actions represent the parallelism or independence that can be exploited in a layer by employing multiple actions at the same time, and vertical relations represent the inherited sequential control flows among the adjacent layers. This hierarchical structure-organization of the heuristics is simple, modular, efficient, and flexible.

Note that the purpose of introducing the hierarchical structure is not to impose a tightly coupled structure into the knowledge base, because not all knowledge can be represented in structured or procedural form. Also, if the structure of the rules is too tight, then the flexibility of the rule-based system may be lost. The purpose of the hierarchical structure is to provide a knowledge organization structure that matches the hierarchical structures in top down problem-solving processes. The hierarchical structure preserves all the advantages of a rule-based system but has better efficiency, modularity, and flexibility in the way it represents knowledge.

An example which applies this technique to the hierarchical decomposition of the parallelism optimization process was presented in [WaGa89]. The implementation of the control in each layer determines the efficiency and effectiveness of the subsystem. One can apply forward reasoning, backward reasoning, or opportunistic reasoning to achieve the best result. By merging the flexibility in opportunistic reasoning of a blackboard architecture and the well-structured control in the *heuristic hierarchy*, we derived a new problem-solving model called the *hier-blackboard*. A *hier-blackboard* is a hierarchical problem-solving model that utilizes the inference power of opportunistic reasoning but follows the control flows inherited from the subproblem decomposition. This achieves a very flexible model that is well-suited to solving complex problems such as optimizing program parallelism.

4.2.4. Opportunistic Reasoning and the Blackboard Architecture

In this section, we will discuss a new problem-solving model called the *hier-blackboard*. The *hier−blackboard* extends the power of the *heuristic hierarchy* by employing the *heuristic hierarchy* for structured organization and dynamic control, opportunistic reasoning for inference, and blackboards for information sharing and communication. This results in a flexible and powerful problem-solving model that is well-suited to complex and ill-conditioned problems such as program parallelization and optimization.

4.2.4.1. The Blackboard Architecture

The *blackboard architecture* [Nii86b, Nii86c] combines the *blackboard* and the *opportunistic reasoning* model. The opportunistic reasoning model [Hayes83, Hayes85] is a problem-solving model in which pieces of knowledge are applied either forward or backward at the most opportune time; whereas the blackboard is a centralized knowledge representation method in which solution states and information are kept in a shared blackboard.

In a blackboard system, the solution space is divided into one or more application-dependent hierarchical levels and is stored in the *blackboard*. Information at each level in the hierarchy represents partial solutions currently known to the level. The problem task domain is divided into loosely coupled subtasks which correspond to areas of specialization within the task. Accordingly, the knowledge for computing intermediate results and performing subtasks is organized into modules called *knowledge sources*. The knowledge sources are logically independent and specialized entities, and they communicate with each other only through the uses of the blackboard. During the problem-solving process, the knowledge sources post information or intermediate results onto the blackboard to update the state of the solution incrementally and they act according to the information in the blackboard. If more than one knowledge source is willing to make contributions, the conflict is resolved by a unit called *control*. The control uses control strategies to choose the most appropriate knowledge source(s) to update the solution state in the blackboard. Opportunistic reasoning is applied within the overall organization of the solution space and task-specific knowledge: that is, which module of knowledge to apply is determined dynamically, one step at a time, resulting in the incremental generation of partial solutions. This problem-solving process is repeated until an acceptable solution is found or the process cannot continue for lack of knowledge or information.

4.2.4.2. Blackboard Systems and Production systems

Several factors distinguish blackboard systems from production systems. First, the knowledge is organized into independent or semi-independent models in the blackboard system, while in production systems all knowledge is represented as production rules. Second, in blackboard systems, the control decision is distributed into the knowledge sources, whereas in production systems the control is sequential.

4.2.4.3. Advantages of the Blackboard Model

The blackboard model has been a favorite choice for solving ill-conditioned or complex problems because of its following properties: modularity in knowledge organization, flexibility in opportunistic reasoning and potential for parallel implementations.

Studies have shown the effectiveness of the opportunistic reasoning in complex and ill-structured problem domains [EHLR80], [Nii86b]. An ill-structured problem is characterized by poorly defined goals and an absence of a predetermined decision path from the initial state to the goal state. In our case, the problem of optimizing program parallelism falls into this category. The blackboard approach requires no *a priori* determined path; the decision of what to apply next is made during the problem-solving process at run time.

4.2.4.4. Weaknesses of the Blackboard Model

The blackboard model has the following weaknesses:

1. *Centralized and global data is a bottleneck for parallel implementation.* All modifications to the blackboard are visible and monitored by all knowledge sources; this can be very difficult to implement efficiently on parallel machines. This problem can possibly be solved by designating private blackboard sections to knowledge sources.
2. *Lacking general guidelines for implementation.* The division and organization of the solution states, solution knowledge, and solution tasks make a great deal of difference in the efficiency, clarity and effectiveness of the implementation. General guidelines in this regard are needed, but it is difficult to come up with such criteria because of the diversity in the different problem domains. Also, the hierarchical structure of the domain knowledge is blurred by the flat structure of the knowledge sources and control.
3. *Expensive to build.* The blackboard model should be used only when its advantages justify the cost of building it.

4.2.5. The Hier-Blackboard Model

In this section, we introduce a hierarchical multi-blackboard model that we call the *hier-blackboard*. The key idea here is to generalize the concept of knowledge sources to map the structure of knowledge sources to the structure of the solution method and localize the interaction between knowledge sources. A commonly used problem-solving method for complicated problems is the *divide-and-conquer* approach in which the problem is divided into subproblems that can be solved directly or be further divided into subproblems until the subproblems can be solved. This achieves a hierarchical level of problem partitioning which is a natural structure for the organization of the problem-solving knowledge. When the task of the system is divided into analytic levels of knowledge sources that implement the subtasks, the knowledge sources can be implemented as a blackboard subsystem if the subtasks they are responsible for are complicated. By applying this methodology recursively to the derived blackboard sub-systems, we derive a model that is logically and structurally tailored to the particular problem at hand.

4.2.5.1. The Framework of the Hier-Blackboard Model

A *hier–blackboard* system consists of the following four types of components: knowledge sources, blackboards, controls and communications.

Knowledge Sources

Knowledge in a *hier–blackboard* system is organized into hierarchically structured knowledge sources according to the problem domain. In our model, a knowledge source is a conceptual unit that interacts with other knowledge sources which share the same blackboard through explicit information updates on the blackboard. In other words, a knowledge source can be a set of rules, a procedure, or a blackboard subsystem if the subtask warrants the creation of such a subsystem.

Blackboards

There is a *master blackboard* which is the blackboard at the top level of the hierarchy that holds the global solution state of the problem. Optional blackboards can be added in the hierarchical tree structure. The blackboards in the sub-blackboard systems are used to hold private entities needed and produced by the local knowledge sources. Each blackboard subsystem works on its private blackboard until global information updates or accesses are needed; in that case, they simply act like other knowledge sources and compete to update the blackboard at the higher level.

Control

For each blackboard in the model, there is a blackboard controller which can be a set of knowledge sources or a separate module that monitors changes on the blackboard and decides what knowledge sources in the system should be executed in case of conflicts. The idea of having a control blackboard ([Hayes85]) can also be incorporated into this model.

Under the supervision of the control, the problem-solving process proceeds through a series of solution cycles. During each cycle, specialized knowledge sources check the blackboard; they self-nominate (if possible) by reporting their possible contributions to a special section of the blackboard that is called the *registration-board*. The control checks the registration board and picks the most appropriate knowledge sources to perform their actions. User-supplied control strategies can be consulted by control to select the "most appropriate" knowledge sources among those who self-nominated. This process continues until the problem is solved or terminates in failure.

As an analogy, the control structure of a hierarchical blackboard system is very similar to the corporate hierarchy of a company. The divisions are connected by tree-structured command channels. Each subdivision can make decisions on its own, but inter-division matters need to be solved by going through their supervisors.

Communication

As the structure of the system is defined, the communication channels between the knowledge sources and the blackboards are defined implicitly. Interaction between the knowledge sources in the same blackboard subsystem is done through the updates of the local blackboard. Inter-blackboard communication is carried out through communication channels that are specified by the system organization. If a knowledge source is implemented as a blackboard sub-system, the communication control in the sub-system will coordinate its

knowledge sources and update the shared blackboard at the upper level under the request of its knowledge sources. The communication in the blackboard hierarchy is done by *messengers* who run up or down one level in the hierarchy to deliver messages between the levels. A messenger is a special knowledge source which monitors a designated area called a *message box* in the blackboard for communication. When certain data in the message box is updated, certain actions are triggered and the messenger delivers the message to the target blackboard. For example, if one blackboard subsystem decides to modify a piece of the global problem state, it gives the message to its messenger for its parent, and the messenger delivers the message to the upper level. The messenger in the upper level updates the information in its blackboard. The update of the information triggers an action which is to update the blackboard one more level up. In this way, the message will be propagated to the master blackboard. Note that the original knowledge source which initializes the chain of update actions may not even know that the information has been sent up the hierarchical ladder since it is only responsible for contributing to its own blackboard. The messenger mechanism helps to make the blackboard sub-systems modular and clean. The blackboard update operations can be implemented efficiently based on the target architecture to minimize the communication overhead.

Primitives for communication between blackboards include access and update of entities of the parent or children blackboards. For update operations, a condition may be sent along with the request. The condition will be checked on the target blackboard with information present in the target blackboard. This last operation is very powerful since the knowledge source does not need to know the state of the parent blackboard system to update information. This conceptual simplicity is accomplished by the messengers who act as representatives for the subsystems to their parent blackboards. This knowledge encapsulation shields the internal operations inside the subsystems and allows them to appear to their parent blackboards as regular knowledge sources. The messenger model we provide here is simple, but powerful, and efficient operations can be defined through this framework. For example, an entity in the message board can be set up so that whenever data is written to it, the data will be immediately sent up to the message board of the parent. In this way, data can be pipelined up the blackboard hierarchy.

4.2.5.2. Issues for Parallel Implementation

Most early research in exploring parallelism of blackboard systems was based on models of target hardware architectures. Examples of parallel implementations of blackboard systems are distributed systems such as TRICERO ([Will85]) and the *distributed vehicle monitoring test-bed* ([LeCo83]), and concurrent blackboards such as CAGE and POLIGON ([Nii86a, Nii86b]). The distinction between the structures of the underlying computational model and the solution model allows the implementation of the solution to be constructed in a more clean and structured fashion. The multiple level structure of the *hier−blackboard* model can be mapped onto an actual hardware by a dynamic or static scheduling procedure. For an unlimited processor model, the mapping is simple, since a processor can be assigned to each basic knowledge source. When this unlimited processor model is mapped to the actual machine that has a finite number of processors, the static mapping will have to be based on the estimated costs, the structure of the *hier−blackboard*, and the locality of the data. Due to the nature of the uncertainty, the dynamic task-allocation scheme may have an edge on static task-allocation.

The actual instructions to update the blackboards can be tuned to the underlying hardware, but this is shielded from users with the blackboard update operations.

4.2.5.3. Simulation of Parallel Hier-Blackboard on Sequential Machines

When several knowledge sources need to share a processor, a special knowledge source called a *scheduler* to control the execution of a set of knowledge sources on a processor can be provided. The scheduler monitors the regions of the blackboard (not necessarily in the same processor) that its knowledge sources are interested in and activates them when appropriate. In the extreme case, all knowledge sources of a blackboard subsystem share a processor, and the scheduler enables a sequential simulation of the parallel model.

4.2.5.4. Comparison with Other Blackboard Models

The *hier−blackboard* is a generalized blackboard model. It inherits all the benefits of the blackboard problem-solving model but provides the following unique advantages over the traditional flat-structured blackboard model:

1. *Better framework for organization of problem-solving knowledge.* The *hier–blackboard* model provides a framework for better structuring of problem-solving knowledge by matching the structure of the knowledge sources with the decomposition of the problem space and the solution methods.
2. *Locality and efficiency.* Localized communication improves both the locality and the efficiency of the system.
3. *Flexibility in utilizing opportunistic reasoning.* Opportunistic reasoning can be applied only at the spots that need the power and flexibility of opportunistic reasoning. Simpler subproblems can be solved by using more straightforward approaches such as rule-based systems.
4. *Flexibility in structural organization.* The structure of the *hier–blackboard* model is very flexible. Its message-passing method can range from the centralized blackboard model to the distributed message-passing provided by the messenger mechanism. Depending on the problem domain, the structure of the knowledge sources can be flat or a complicated multiple level hierarchy. The flexibility in the structure and the specialized knowledge sources that is built-in make problem solving easier than in the traditional blackboard systems.
5. *Higher potential to be parallelized.* The *hier–blackboard* model is designed with concurrency in mind. Higher locality means higher potential for parallel implementation. Built-in knowledge sources such as the scheduler, control, and the messenger simplify the implementation of the parallel problem-solving model. Depending on the structure, the *hier–blackboard* model can be applied at different degrees of parallelism ranging from sequential to highly parallel.

4.2.5.5. Applying the Hier-Blackboard to Parallel Compilers

To apply the *hier–blackboard* model to parallel compilers, we decompose the process of optimizing program parallelism hierarchically based on the method we described in section 4.2.2. As can be seen from figure 4.1, the most complicated subproblems are the parallelism matching process and the five subproblems for improving general parallelism, creating tasks, allocating processors, minimizing synchronization, and optimizing memory access. These subsystems can be implemented as blackboard subsystems with the parallelism matching module managing the other five blackboard subsystems corresponding to the five subproblems for improving program parallelism. The topmost layer, the parallelism matching control layer that controls the performance evaluation and the selection of the focuses, can be implemented as the control module for the blackboard system for parallelism matching. We encapsulated each program transformation technique into an object that contains the procedure for performing the transformation, the applicability test, heuristics for utilizing the transformation for different purposes for different kinds of programs and target architectures, and heuristics for selecting appropriate arguments to apply (methods of application). These modules form the knowledge sources for the subsystems for improving general parallelism, creating tasks, allocating processors, minimizing synchronization, and optimizing memory access. We also chose to implement the program parallelism analysis and machine parallelism analysis as rule-based systems that can be activated by the program parallelism matching subsystem and its knowledge sources.

4.2.6. Comparison Of The Three Frameworks

We discussed three different frameworks for realizing the program optimization process for parallel computers. We will compare these three frameworks from five aspects: optimism, efficiency, flexibility, simplicity and parallelism.

The *hier–blackboard* is an extension to the *heuristic hierarchy* so they share most of the characteristics except that the *hier–blackboard* utilizes the parallelism within the program optimization process and features opportunistic reasoning which is much more sophisticated and flexible than the control in the *heuristic hierarchy*. The opportunistic control in the *hier–blackboard* model is a bit more complicated than the rule-base approach used in the control on the *heuristic hierarchy* and is more difficult to program (but it is still significantly simpler than the general blackboard architecture due to the modular organization and abstraction).

The A^* algorithm guarantees that the resulting solution will be optimal as long as the heuristic function h is a lower bound of the actual cost h^*. The other two approaches use heuristic rules to select the transformations and thus lose the optimism. The efficiency of the A^* algorithm depends on how close the heuristic function h is to h^*. In other words, the A^* algorithm can be very efficient if the performance estimation function h is very accurate. However, accurate performance estimation at compile time is as hard as the program optimization issue if not any harder. This means that a compromise between the accuracy of the performance

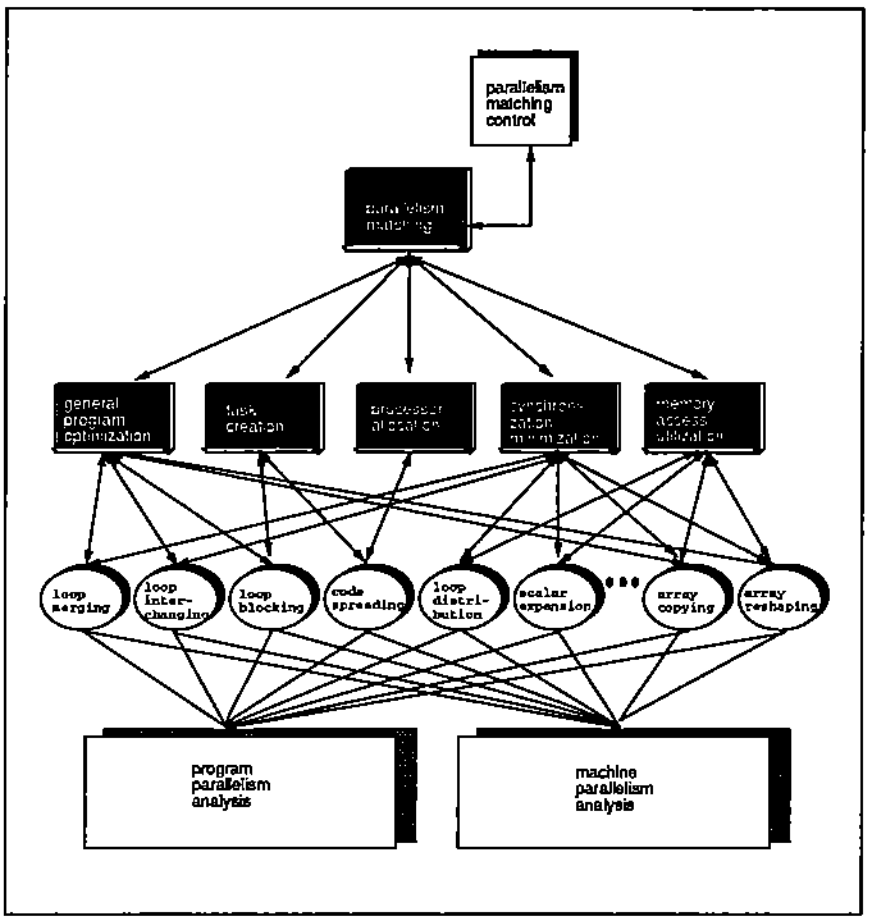

Figure 4.2. *The structure of a program parallelism optimizing system based on the hier−blackboard model.*

estimation and the efficiency of the program optimization process is needed.

The efficiency of the hierarchical reasoning framework depends heavily on the richness of the knowledge. AI techniques such as pattern recognition, constraint propagation, goal reduction, and learning can be used to improve the efficiency of the system.

The A^* algorithm is not as flexible as the other two approaches since the heuristic functions are applied to the whole decision tree, while the decision-making processes of the other two approaches can have different control strategies at different stages.

The above comparison is summarized in table 4.1 below. Our original implementation of the prototype system used the *heuristic hierarchy* [WaGa89], but was later transferred to the *hier-blackboard*. We also studied the A^* algorithm because of its optimal characteristic. The selection of the framework is based on many complicated considerations and does not imply that any of the framework is significantly better than the others.

Table 4.1. *A subjective comparison of the three frameworks: the A^* algorithm, the heuristic hierarchy, and the hier−blackboard.*

	optimism	*efficiency*	*simplicity*	*flexibility*	*parallelism*
A^* algorithm	*optimal*	*depends*	*good*	*fair*	*NA*
heuristic hierarchy	*depends*	*good*	*fair*	*good*	*NA*
hier−blackboarddepends	*depends*	*complex*	*no*	*excellent*	*excellent*

4.3. Conclusion

In this chapter, we discussed three frameworks for implementing the feature-directed program optimization framework in a parallel compiler. The three frameworks are discussed in detail and are compared based on the optimism, efficiency, simplicity, flexibility, and parallelism. We will discuss issues that support the framework, such as machine knowledge manipulation, performance prediction, heuristics and transformation techniques to improve program parallelism, in the next three chapters. In chapter 8 we will describe a prototype intelligent parallel programming environment that are implemented based on the *hier–blackboard* framework.

CHAPTER 5
MACHINE KNOWLEDGE MANIPULATION ISSUES FOR PARALLEL COMPILERS

5.1. Introduction

In this chapter, we study issues related to the representation and manipulation of the knowledge of machine parallelism and the implication of these issues in parallel compilers. The main questions that we are concerned include what kind of machine knowledge a parallel compiler needs to have, how to analyze program optimization heuristics and identify machine features involved in them to describe the knowledge, how to represent the machine knowledge and the heuristics, how to acquire the inference capability, and how to support different classes of parallel computers.

An object-oriented hierarchical machine feature representation scheme that is designed to address these problems is presented. This representation scheme features modular knowledge representation, various degrees of abstraction, and hierarchical reasoning. It provides a foundation for systematic analysis of heuristics and allows parallel compilers to support different classes of target architectures. A machine knowledge manipulation system that realizes the representation scheme is also presented. The system is implemented in Prolog and provides a mechanism for interactive machine feature specification and classification; it also supports reasoning based on the program and machine features.

5.1.1. Feature-Directed Program Optimization

As discussed in the last chapter, methodologies for optimizing parallelism on a particular parallel machine are often based on features of the architecture. Under the *feature-directed program optimization model*, optimizing parallel compilers use the features of the program and machine *explicitly* to control the restructuring of the programs. Unlike other parallel compilers where decisions for program optimization are based on the implicit heuristics that are hardwired and scattered in the compilers, this approach allows the compiler to base its decisions on features of both the target machine and the program. In this way, the compiler is actually "*programmed*" by the features of the chosen target machine and the program to be optimized. For example, figure 5.1 shows the flow graph of a simple heuristic for loop blocking that is based on machine features.

The effectiveness of knowledge-based compilers and multiple target parallel compilers relies on a suitable knowledge representation and processing schemes for representation and manipulation of the machine parallelism and program optimization knowledge. The machine knowledge representation scheme needs to provide a foundation for the integration and organization of the program optimization knowledge and support for performance evaluation and reasoning. In section 5.2, we discuss machine features that are important to parallel program optimization and the requirements for knowledge representation in parallel compilers. In section 5.3, we present an object-oriented machine knowledge representation scheme which features modular knowledge representation, various degrees of abstraction, and support for hierarchical reasoning.

Recognizing and collecting useful heuristics and analyzing and separating machine features from the program optimization heuristics are very involved jobs. A good knowledge manipulation system can help knowledge engineers comb through the complex and ill-organized knowledge and identify the essential elements of the knowledge to help them transform fragmented heuristics into well-defined programs. Automatic tools to help knowledge engineers to program parallel compilers are highly desired and are long overdue. In section 5.4 we introduce a machine knowledge manipulation system that is based on the knowledge representation scheme discussed in section 5.3. The machine knowledge manipulation system can be used to interactively analyze heuristics for optimizing parallelism, comparing machine features, abstracting new machine

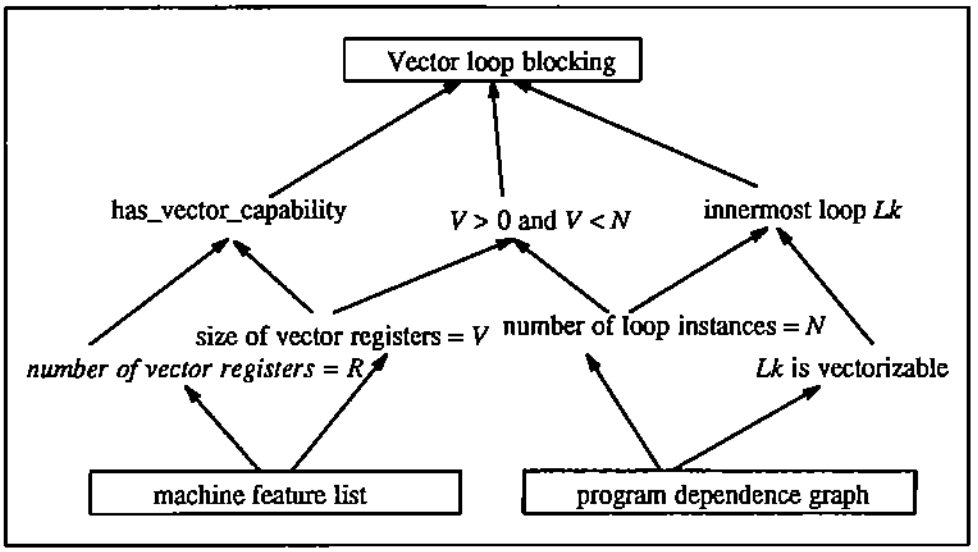

Figure 5.1. *A simple example to illustrate feature-directed program optimization.*

knowledge, and installing new target machines.

5.2. Machine Features and Parallel Compilers

5.2.1. Machine Features

Properties of the target machine that affect the concurrent execution of the machine are called *machine features*. Definition of the machine features records the distinct properties of the architecture that need to be considered in utilizing the parallelism on the architecture. The manipulation of machine knowledge includes representation and organization of machine features, inference and modification of machine features, and support for reasoning based on machine features.

A detailed discussion of machine features and their effects on program transformation was presented in [WaGa87]. In this section, we examine different aspects of machine features (such as interaction between features, abstraction and classification of the features) and the criteria that these aspects impose on the representation of machine features.

5.2.2. Important Machine Features for Parallel Compiler

Functionally, a parallel computer can be characterized into four components: the processing elements, the memory hierarchy, the communication networks, and the control unit.

The processing element is the hardware unit that carries out the computations. The memory hierarchy is a functional unit that provides data stores of different speeds. The communication network interconnects the different components of the system. The control unit is the functional device that controls the execution of the parallel computer. The processors can also be grouped into clusters. A *cluster* is a collection of processors that is capable of executing a collection of tasks in a tightly coupled manner. For example, the computational complex (CEs) of the Alliant FX/80 forms a cluster that is distinct from the interactive processor system in the FX/80. A system may support multiple clusters with multiple processors per cluster, as in the *Cedar* system [KDLS86], or it may be viewed as one tightly coupled cluster of processors, as in the Connection Machine, or a loosely-coupled system of one-processor clusters, as in the Cray XMP.

A parallel computer may have some or all of the above components. Since there are so many possible combinations of the machine configurations, it is impossible to describe all cases here. Instead, we will study variations of some important features and their effects on the programming methodologies and the parallelism optimization issues of parallel computers.

5.2.2.1. The Processing Elements

A processing element may contain one or more scalar arithmetic processors and vector processors. A scalar arithmetic processor processes data sequentially and can be characterized by the costs of basic operations and the number of functional units (adders, multiplier, etc). Parallelism which exists in this level is in very fine-grain and has been explored extensively in sequential compilers.

The vector processor contains one or more vector pipeline(s), vector registers and vector operation control. The vector pipeline overlays the execution of the operations, and can be characterized by the number of pipelines and the number of stages in each pipeline.

The vector operation control unit controls the loading, storing, chaining and execution of the vector pipelines. It can be characterized by the following features: vector instruction types, costs of vector operations, vector startup time, vector operation chaining, cost of non-uniform stride operations, cost of scatter-gather, and vector reductions. For a machine that has vector registers, it is important to keep the vector operands in the registers and to match the length of vector operations with length of vector registers. For a machine that has no vector registers, the operands of the vector operations need to be in the memory (e.g. Cyber205). Important issues for these kinds of machines include increasing vector lengths, avoiding bad strides, and chaining the vector operations.

Table 5.1. *Machine features of processing elements of some parallel computers.*

Feature Name	nCUBE 2	iPSC/860	Alliant FX/8
Number of processors	64	512	8
Maximum number of processors	8196	1024	8
computation mode	parallel	parallel	parallel+vector
scalar instruction type	three-address	three-address	three-address
scheduling method	[user control]	[user control]	[user_control, self_scheduling]
clock cycle rate	20	40	
peak performance MIPS	7	40	10
peak performance FLOPS/PE	3.5	80	11.75
peak performance / vector pipe		40	30
address bits	26	32	28
operation overlapping	[data_prefectch]	[alu_fpu_units, fpu_fpu_units]	[alu_fpu_units]
fpu-fpu operlapping operations		[add-mul]	
vector capability	no	no(floating-point pipeline)	true
vector startup time		small	small
vector chaining			true
vector operand			register
vector reduction			[inner-product, min, max, ...]
number of pipelines(per proc.)		2	1
pipeline stages		3	5

5.2.2.2. The Memory Hierarchy

A complete memory hierarchy may have global memory, cluster memory, local memory, cache memory, and vector memory. Most parallel computers utilize two or more levels of memory hierarchy, but some have only shared or private memories.

Global memory can be accessed by all processors, and can be either physically centralized in one memory module (as in the Alliant FX/8) or distributed among processor units (as in the BBN Butterfly and the IBM RP3). Cluster memory is shared by the processing units in the cluster (e.g. CSRD Cedar). Local memory is owned exclusively by individual processors. However, some computers have a centralized controller which can access all local memories (as in the Pringle [KGSF84, KWGCS84], or the Connection Machine [Hillis85]). Another extreme is the memory modules in the IBM RP3 where private and shared memories share the same module and the boundary between them is shiftable and controlled by software. Cache memory is usually a very fast memory module; there is usually a high bandwidth bus between memory

and cache so that data can be transferred in blocks. The vector registers are used to store the vector operands so that memory accesses can be minimized for vector operations. Important features about vector registers include the number and the size of the vector registers and the cost of loading and storing information into vector registers.

Important features about the structure of the memory hierarchy include the size of different memories, memory sharing (shared cache, private cache, share memory, private memory, etc), cache coherence strategy (compiler management, snooping cache, etc), centralized or distributed memory modules, memory interleaving, cost ratios of different memory accesses, vector pre-fetch mechanism, and available memory synchronization commands (fetch-add, locks, memory tags, etc).

The major goal of memory management is to minimize the data access time. This can be achieved by removing unnecessary data dependencies, keeping data in the fastest memory, changing memory access patterns, overlapping memory accesses with other operations, using block memory accesses, deciding what data to cache, and so on.

Table 5.2. *Machine features for memory hierarchy of some parallel computers.*

Feature Name	nCUBE 2	iPSC/860	Alliant FX/80
number of vector registers		0	8
size of vector registers			32
data cache size (bytes)	0	8000	128000
cache block size(bytes)		32	32
cache coherence strategy		write_back	write_back
shared cache	no	no	yes
local memory size (Mbytes)	4	16	0
max local memory (Mbytes)	64	4000	0
global memory size (Mbytes)	0	0	64
maximum global memory(Mbytes)	0	0	128
vector prefetch emchanism			['global to cache', 'global to register']
special synchronization instr			['memory tags']
register/cache access cost ratio		2	2
cache/memory access cost ratio		3	5

For machines with only global memory with uniform access times, the primary issues are minimization of synchronization and critical regions and utilization of fast registers.

For machines that have local memory, cache memory, or global memory with non-uniform access times, locality of the data becomes very important. In such situations, deciding where to put the data is generally based on the ratios of the different memory access times. Storing shared data in the local or cache memory introduces the data coherence problem along with the overhead of moving the data. Only when the gain in shorter accessing time outweighs the loss in overhead of transmitting and updating of the data should a local copy of the data be created. Some machines support block access of the data which can decrease the data transmission time and should always be used if possible. For a machine that has only local memory, message-passing strategies are the basis of all synchronization and accesses to shared information. Issues to be considered include domain decomposition, load balancing, ratio of communication and computation, and overlapping communication with computations. On most networks, the unit cost of transmitting data decreases as the message gets longer. Therefore, long messages are usually favored. On the other hand, long messages need more time for computing so may undesirably increase the synchronization cost. Striking a balance between the communication and computation costs is necessary and is non-trivial.

5.2.2.3. Interconnection Networks and Busses

The connections between components of the system (processors, memory modules, clusters, etc) can be either busses or more complex interconnection networks. A bus architecture has the advantage of simple and fast data transmission, but it allows only a small number of devices to be attached to it. For machines with busses, the primary concern is the cost of memory accesses. For machines with network connections,

important features of communication networks include network topology, network bandwidth, delay per network stage, packet or circuit switched, packet size, maximum number of pending memory references a processor can have in the network, network routing (self-routing, store-and-forward, worm-hole-routing, etc), and the performance penalty of self-routing.

Network topology is probably the most important property of a network. It affects the algorithms for solving the problem and the way the data are distributed in the system. On networks with low bisection widths, certain data movements are notoriously slow. For example, a matrix transpose is extremely costly on trees and rings. A complete study of the role of topology in parallel algorithm design is found in [GaVR84].

From the compiler's point of view, there are two critical issues in network management: the routing algorithms and minimization of network traffic. Some computers have self-routing hardware or software for which routing mechanisms like worm-hole routing ([Dally87]) can save significant data transmission time and ease the job of both programmers and compilers. For other machines such as the Pringle, the compiler needs to plan a path and generate code to perform the routing. Also, If the network is such that some processors are "nearer" than others, and if the message delay from a far processor is significantly more than from a near processor, optimal data placement becomes critical. Unfortunately, not only is this problem NP-complete, but there are also very few good heuristics for it. Techniques to solve this problem include minimizing intersections of subdomains, minimizing distances the messages need to travel, accumulating data into longer messages, and using code motion to minimize synchronization delays.

For machines that support self-scheduling loops, the program restructuring system can leave the task scheduling problem to the operating system of the machine at run time by changing the outermost loop into a self-scheduling loop. However, using the self-scheduling loops makes good global array decomposition almost impossible, since it can only be known at run time which loop will be run by which processor. For machines that have no combining network, balancing the computational load and avoiding network contention are among the major challenges to parallel compilers.

Table 5.3. *Machine features for communication networks of some parallel computers.*

Feature Name	nCUBE 2	iPSC/860	Alliant FX/80
interconnection type	network	network	network/bus
network topology	hypercube	mesh(16x32)	crossbar
memory-cache interconnection	channel	channel	bus
cache-PEs interconnection	NA	bus	crossbar
memory-cache bus bandwidth			150 MB
cache-PE network bandwidth			376 MB
network bandwidth (Mbytes/channel)	5.5	25	47
number of channels per processor	14	4	
message start up init α^0, μs	158.45	100	
message start up per hop α^1, μs	1.29	1.0	
message transmit per word β^0, μs	2.49		
min package size (bytes)	0	100	1
routing control	[self-routing]	[self-routing]	NA

5.2.2.4. The Control Unit and Processor Clusters

Some machines have control processors that control the execution of the processing units (for example, most SIMD machines have a single control processor). On other machines, the control exists only in the form of cooperating software at the operating system level. On machines with centralized control at the instruction level, such as SIMD machines, VLIW machines, synchronization overhead is small, so parallelism can be utilized at the fine-grain (instruction) level. In particular, for VLIW (very long instruction word) machines, all scheduling, runtime synchronization and communication are completely specified at compile time. This results in very expensive compilation and limited parallelism. Opposite to the VLIW machines are the data flow machines where processors are activated by the flow of the data. The control on a dataflow machine tends to be very fine grain and dynamic. The major optimization issue for dataflow machines is to minimize the depth

of the dataflow graph of the program. This can be done by common subexpression elimination, code motion, dead-code elimination, constant folding, etc.

Another type of control involves dynamic task scheduling on most shared memory MIMD systems. This kind of system usually utilizes the parallelism at the process level with dynamic process control. Depending on the configurations of the machines, the cost of process initialization and process switching may vary from very cheap (light-weight task) to very expensive (heavy-weight process). For machines that support light-weight processes (tasks), the major issues are the scheduling of tasks and the minimization of communications. For machines that do not support light-weight tasks, large processes need to be created. This usually results in matching the number of processes with the number of processors.

For machines with multiple clusters, there are two levels of scheduling: a "micro-task" level that manages jobs within each processor and a "process" level that assigns processes to each cluster. The major issue to consider is the granularity of the tasks and locality of data in the clusters.

Features of the control unit at the hardware level include the number of processors, hardware primitives for scheduling (for example, the fetch-and-add operation), types and features of shared resources, task scheduling time, process switching cost, and primitives for scheduling and their costs. The control features at the operating system level include the scheduling mode (self-scheduling, data-driven, etc), scheduling heuristics and system scheduling operations. At the programming level, the control includes user assertions, heuristics for process scheduling, explicit parallelism control constructs, and timing specification for real time control programs. Features at the cluster level category include cluster size, homogeneous or heterogeneous clusters, shared resources within clusters, task switching time within a cluster, and processor scheduling policy within cluster.

5.3. Design Considerations for Machine Knowledge Representation Schemes

The machine features listed in the last section represent fragmented knowledge of the machine and lack a comprehensive understanding of the whole architecture that they describe. In order for the system to build up an understanding of the target machine, the knowledge needs to be connected by relationships between the features and the knowledge of how these features affect the parallelism of the target machine. This implies that the knowledge representation will have to support the composition of the knowledge and mold pieces of the knowledge together to obtain a whole view. To avoid a tedious specification process, the task of finding relationships between features should be done only once for each pair of features and not repeated for other machines. In the next section we will describe an object-oriented knowledge representation scheme which allows one to build up the structure among the features. Before we get into the details of the representation scheme, we discuss several design decisions for supporting the reasoning capability of the feature-directed program restructuring model. Should the machine knowledge base be organized as a flat-structured feature list or a hierarchical-structured tree? Should the representation allow implicit knowledge implication or should all relationships be spelled out explicitly? These decisions affect both the power and elegance of the representation scheme.

Flat Structure Versus Hierarchical Structure

Representing the machine features in a flat structure has the advantage of being simple and explicit. In [WaGa87], a flat-structured, uniform machine feature representation scheme is used to represent and model parallel computers. All basic features are represented as facts in the database. When a target machine is specified, the features of the machine are loaded into the kernel of the expert system. The features are abstracted by applying a set of rules to the facts. This approach is simple and yet powerful enough to model the parallel architectures. Specification of the machine can be done feature by feature and is straightforward. However, the flat-structure representation does not have a mechanism to show the relationship and interaction between machine features. The relations between the features must be encoded by using other constructs such as the rules and pre-conditions used in the above system. To find the relationship between particular features, one must exhaust the rules to find the rules that define the relation between them. This makes manipulation and maintenance of the system an involved task.

A hierarchical knowledge representation scheme incorporates the interrelationship into the representation of the knowledge itself. The relationships between the machine features are mostly architecture independent and can be inherited from the previous known structures. Therefore, they do not need to be redefined when a new target architecture is defined. On the other hand, it is important to have the ability to define new

relationships so that similar architectures can share most of the knowledge but are allowed to specify the differences explicitly.

To effectively support the hierarchical knowledge representation, the knowledge representation scheme should support abstraction and classification. Classification allows grouping of relative features into distinct classes that possess special features, whereas abstraction defines relationships between features of different levels. For example, figure 5.3 (b) on page 125 shows an organization of the machine features; the vertical dimension shows the abstraction levels of the architecture and the horizontal dimension shows how related features are grouped together to form the base of the abstraction. Together, classification and abstraction provide a powerful mechanism for the integration of feature knowledge. Our representation scheme defines a hierarchical structure by using the relationship between the subclass and superclass along with relationship functions.

Explicit Versus Implicit Knowledge Representation

Knowledge of the machine can be explicitly expressed or implied by other knowledge. Explicit knowledge has the advantage of being simple and immediately available, but may contain redundant knowledge and inflate the size of the knowledge base. On the other hand, allowing implicit knowledge representation leads to more concise representation but increases the difficulty of maintaining the knowledge base. To balance the tradeoff, it is a good idea to make all knowledge implication rules explicit.

For example, in the following example, the feature *cost ratio of global memory access and computation* is defined to be the ratio of the cost of a global memory fetch and the cost of a floating point multiply. This property of the machine can be defined explicitly or implicitly by the features *cost of a global memory fetch, cost of a floating point multiply,* and rule 5.1 which explicitly defines the relation.

Rule 5.1 feature_value('cost ratio of global memory access and computation', R) :-
feature_value('cost of a global memory fetch', F),
feature_value('cost of a floating point multiply', M),
R is F / M.

The advantage of spelling out the relationship instead of specifying the value for the information is that when the feature is to be modified (for example, when the memory is upgraded into fast memory), the feature *cost ratio of global memory access and computation* does not need to be redefined. On the other hand, when optimizing a program for a particular machine, only the relative cost ratio will affect the decisions. So when the actual costs of memory access and computation are not required elsewhere in the system, the value of the ratio can be specified explicitly.

5.4. An Object-Oriented Knowledge Representation Scheme for Parallel Computers

From our point of view, an intelligent parallel compiler needs a "simple" machine knowledge representation scheme that can support reasoning, knowledge abstraction, organization, and heuristic comparison. The scheme we describe here is based on an object-oriented knowledge representation paradigm, in which features about machines are treated as objects and the understanding of a machine can be composed from feature objects.

The object-oriented paradigm is a sound knowledge representation methodology. Its modularity and hierarchical abstraction capability make it particularly appealing for the representation of complicated real world knowledge. The inheritance and chaining of this paradigm allows a compact and elegant representation of the relationships between objects; this also makes porting the system to new machines easy. On the other hand, just like any other knowledge representation paradigms, there are many tradeoffs in the object-oriented paradigm and design decisions in the implementation that must be made based on the application domain. In this chapter, we concentrate on just one application domain -- multiple-target parallel compilers -- although most of the principles and methodologies we have discussed here can be applied or extended to other problem domains as well.

Our knowledge representation scheme consists of three elements: the feature objects, relationship between objects, and operators on the objects. Under this scheme, the knowledge of the parallel computers can be decomposed into features and reassembled into hierarchical models based on feature organization and abstraction techniques. This knowledge representation scheme provides a vehicle for heuristic manipulation and intelligent compiler construction.

5.4.1. Feature Objects

We followed the convention of the object-oriented paradigm by which objects representing machine features are represented by a set of basic objects that we call *feature objects*. Feature objects are classes of objects that are the basic units for defining properties of parallel computers. A feature object represents a particular property of the target architecture and has slots to store information about the property. These slots are called the attributes of the feature object. The structure of the feature object varies by the type of the property it represents. Some attributes of the features are common to all features and some apply only to certain features. The template of feature objects is defined by a meta-class that we call a *feature class*.

Each entry in the feature class defines possible values for feature objects. An instance of a class is a particular instantiation of a class. An instance of the feature class identifies the properties of a machine feature. These properties include the name of the feature, type of the feature (used for type checking), conditions for this feature to be meaningful, relation of the feature with other features, and the attributes of this feature. The feature object, which defines what the instances of the feature should be, is the foundation of the representation. The example shown in figure 5.2 defines a feature named *vector registers*. In the example, the feature object *vector registers* is active only when the target machine has vector capabilities. This object is a subclass of the objects *register* and *memory*, so it inherits all properties of the two classes. Since all machine features are subclasses of the meta class *feature object* the latter can be omitted from all definitions.

In figure 5.2, slots in the first part of the template are common for all feature objects (inherited from the class *feature object)*. Slots in the second part are the attributes of the feature.

A class describes the implementation of a set of objects that all represent the same kind of knowledge. The individual objects described by a class are called its instances. An instance of a feature object is defined when a value is associated with the object by a feature assignment statement: *featureDefine*. For example, the following Prolog function causes a message to be sent to the object *vector_registers* and sets the values of the *number* and the *size* of the object to be 64, 32, respectively.

featureDefine(vector_registers, number, 64)
featureDefine(vector_registers, size, 32)

When defining the machine features, the following alternative form is also accepted:

the number of vector_registers is 64.
the size of vector_registers is 32.

The statement *featureValue* provides a uniform way of accessing the feature instances. For example, *featureValue(vector_registers, number, N)* returns the value of the current instance of the *number* of the feature *vector_registers* in variable *N*.

5.4.2. Attributes Associated with Objects

New attributes can be associated with a feature object through the feature-attribute assignment statements:

class *vector_registers* **subclass_of** *register and* **subclass_of** *memory* **with**
pre-conditions: has_vector_capabilities,
static_dynamic: static,
type: [boolean, true],
number: [integer, 64],
size: [integer, 32].

feature name:	vector_registers	
pre-conditions:	has_vector_capabilities	
static/dynamic:	static	
relations:	parent(vector_capabilities)	
feature type:	boolean	
feature value:	true	
attributes:	type	value
number of vector registers	integer	64
size of vector registers	integer	32
other attributes:	omitted	

Figure 5.2. *The definition of the feature object vector_registers.*

featureAttribute(ObjectName, AttributeName)
featureAttribute(ObjectName, AttributeName, Type, DefaultValue).

The first statement defines the relationship between the attribute and the feature, and in the second statement both the relationship and the value are defined. A feature attribute assignment statement explicitly and dynamically defines the binding between the feature object and its properties.

A object defined to be dynamic may have more than one instance but only one instance can be *current*. In some cases, allowing more than one instance of the same object provides a degree of "non-determinism." This is useful when the target machine contains multiple characteristics. For example, on hypercube computers, the communication network can simulate different kinds of networks such as rings, meshes, shuffle-exchange networks, and trees. Different algorithms can utilize any one of the communication patterns. On the other hand, the current instance mechanism provides a way to obtain the deterministic effects.

5.4.3. Feature Organization

Feature objects form the basis of the representation of parallel computers. In the real world, knowledge of objects is not isolated. Instead, relationships exist between knowledge on different levels of abstraction. Therefore, representation of the relationships between objects by using a higher level of abstraction and organization is needed so that the relationships can be recorded and manipulated. The bottom-up approach allows feature objects of similar properties to be grouped to form a feature object of higher level. Using a top-down approach, one may decompose the feature objects into more detailed descriptions and continue expanding until the feature is a basic fact. With either approach, the feature classes are organized into hierarchical structures based on *relation functions*. The relation functions explicitly define the relation between feature objects of different levels. This includes predefined relations such as parent, children, exclude, complement, associate, compose, along with user-defined relation functions. The relation functions serve two purposes. First they define the relationship between objects; second, they define the flow of control and messages. The collection of the relation functions organizes the machine knowledge into a hierarchy of features. Some of the relationships, such as parent and children in the class hierarchy, are inherited from the organization of the hierarchy and are defined by the superclass or subclass_of relations; others need to be explicitly defined.

The organization of the features can simplify the representation of the features. For example, the pre-condition 'has_vector_capabilities' listed in the example in figure 5.2 is redundant because the feature *vector registers* is the descendant of the feature *vector capabilities*, and this pre-condition is actually implicitly encoded in the hierarchy of the features.

5.4.4. Operations on the Objects

5.4.4.1. Inheritance, Specification and Qualification

The notion of classes and meta-classes provides a mechanism for sharing information between different objects via inheritance. *Inheritance* allows a subclass to inherit its properties from a superclass. For example, defining the class *local_memory* to be a subclass of the class *memory* implies that *local_memory* inherits all properties of *memory* unless otherwise specified.

Different subclasses or instances of a class can have distinct properties through the use of *specification*. For example, *cachememory* is also a subclass of *memory*, but it has the property *cachecoherencescheme* that the class *localmemory* does not have.

A class can have more than one superclass, and it inherits all the properties of its superclasses. For example, the class *vector_register* defined above is a subclass of both *memory* and *register*.

Inheritance and specification are used to group closely related knowledge into the same class so that the information can be accessed locally in the system. Specification helps to distinguish objects in the same class, while inheritance keeps the size of the system manageable.

Inheritance and specification are usually associated with *qualification*. That is, an inheritance or specification for an attribute is applied to an instance of a class when certain conditions are satisfied. For example, in distributed computing, achieving a balance between computation and communication is important. Also communication with processors that are far away is generally more expensive. These heuristics can be inherited by knowledge of all distributed computers.

```
class 'distributed memory' subclass_of 'global_memory'
    .....
    heuristics: [heuristic('balance computation and communication ratio'),
                 heuristic('avoid far access'), ...],
    .....
```

However, when the cost ratio of far-access and near-access is close to unity, sending messages to far away processes does not incur a significant penalty. In this case, the restriction can be lifted by changing the attribute 'avoid far access' to 'far access OK' as below. In the following example, the heuristic 'far access OK' is a specification that overwrites the inherited heuristic 'avoid far access'; and the conditions in the qualification statement *in_case* validate the specification statement.

```
'distributedMemory' instance_of 'distributed memory' with
    .....
    heuristics: [
      heuristic('far access OK',
        in_case (featureValue(distributedMemory, 'far/near access ratio', A),
                 isSmall(A-1.0)))
        ..... ],
    .....
```

5.4.4.2. Feature Modification

Some machine features can be changed during the course of program optimization. This provides flexibility in both matching the algorithm with the machine and reasoning. We call features that can be modified *tunable* features, and features that cannot be changed are called *static* features. Possible modifications to a feature object include updating the feature value, changing the feature attributes, and modifying the heuristics associated with the feature. An instance of a feature object can be modified by the *featureDefine* statement that we described above. For example, suppose the target machine is a hypercube computer. Since the hypercube can simulate other network topology (like mesh, trees, and shuffle exchange networks), at certain stages of the algorithm, a particular type of communication topology may match the algorithm better than others. The communication pattern can be easily changed by the following statements:

featureDefine(networkTopology, network, mesh, OldTopology).
..... communication using mesh .....
featureDefine(networkTopology, network, OldTopology, _).
..... communication using OldTopology .....

The feature modification operation can only be applied to the *tunable* properties of the architecture and attempts to modify static features will fail.

5.4.4.3. State Adjustment with Dependencies

One problem that arises from object modification involves maintaining integrity and consistency. In our object-oriented scheme, this problem is addressed by providing a dependence-mechanism to notify an object of changes in related objects. The relation functions explicitly define the dependence relations between the objects involved. When the source of the dependence is changed, the target object is notified. And the target object may examine the change to decide whether to change its own state or not. For example, the heuristic object *utilizing vector registers* depends on the feature objects *available vector registers* and *vector operations.* If the object *available vector registers* is changed and no vector register is available, then the object *utilizing vector registers* needs either to move certain data in vector registers to memory to reclaim the vector register for the next operand or get the next operand from the memory. What action to take depends on the heuristics in the heuristic object.

5.4.5. A Simple Example

The specification of a feature involves defining the template (class) and the value (instance) of the object. The former is normally the task of the system implementor and the latter is the task of system maintainer who installs a new machine. For example, the following is a fragment of the program that specifies the template for the global memory in a shared-memory architecture.

```
class memory_hierarchy with
        type: one_of [shared,distributed,hybrid],
        structure: list_of [global,cluster,local,cache].
class memory with
        type: one_of [shared,distributed,hybrid],
        size: integer,        % 1 unit = 1K bytes
        ratio_of_fetch_and_multiply_op: real,
        ratio_of_fetch_and_register_fetch: real,
        interleave: [integer, 4],
        read_cost: real,   % in clock cycles
        write_cost: real,   % in clock cycles
        prefetch: [boolean, false],
        prefetch_size: integer in case prefetch == true,
        connection: one_of [bus, network], % to processor
        network_topology: one_of [hypercube, omega, crossbar, ...]
                in_case connection == network.
class local_memory subclass_of memory with
        type: local,
        size: [integer,4000],      % specify default value
        connection: bus.
```

Under the above definition, the local memory on a nCUBE 2 can be defined as:

memoryHierarchy instance_of memory_hierarchy with
type: distributed,
structure: [localMemory].

localMemory instance_of local_memory with
size: 4000 in_case pid < 16, % has uneven memory distribution
size: 1000 in_case pid >= 16,
ratio_of_fetch_and_multiply_op: 0.14,
ratio_of_fetch_and_register_fetch: 3,
prefetch: true,
prefetch_size: 8,
connection: bus.

The entries can be accessed or updated in the following way:

featureValue(localMemory, size, K).
the size of localMemory is K.
the connection of localMemory is bus.

An interactive feature-specification scheme is described in section 5.6.2 as an alternative way to interface with the machine feature manipulation system. Different types of people involved with the system (users, system implementors, or system programmers) can use different interface schemes.

5.4.6. Features of the Parallel Machine Knowledge Representation Scheme

The most important feature of our knowledge manipulation scheme is in the flexibility and the modularity of the scheme. The representation scheme can be used to achieve the following features.

1. Allows dynamic modification of machine knowledge.
2. Has high flexibility in knowledge characterization and organization.
3. Has various abstraction levels of the knowledge.
4. Supports knowledge hiding and global visibility.
5. Supports inference and knowledge encapsulation.

5.4.6.1. Static and Dynamic Knowledge Representation

Although most features of a parallel architecture are fixed after the configuration of the system is set up, some features of the machine should be allowed to change dynamically. Allowing dynamic modification to some machine features has the following advantages. First, conceptual decomposition of the machine is possible. This means that the programmer can decompose the computational model to match the algorithm decomposition such as divide-and-conquer algorithms. Second, users are allowed to define "virtual machines" based on the existing machine features and knowledge. This is ideal for testing new machine designs or programming heuristics. Third, computer vendors usually provide many different configurations to suit specific needs of the users. Fourth, specification for similar architectures can be more elegant and compact. The representation scheme we defined above allows the representation of both static and dynamic knowledge in a uniform structure. Machine features that can be modified at run time are marked as dynamic features by setting the attribute dynamicFeature to be true. Dynamic machine features include the number of processors used in computing, system load, algorithmic network topologies, and task control strategies. Features that are static over time are termed static features and attempts to modify static features will be rejected by the system.

5.4.6.2. Flexibility in Knowledge Characterization and Organization

Knowledge characterization is the basis for knowledge organization. Machine features can be characterized in different ways under different constraints. For example, features of a parallel computer can be categorized by the physical component modules (such as processing units, memory modules, communication medium, clusters, and control) or by the functionality of the features (such as program partitioning, data decomposition, process scheduling, memory utilization, communication minimizing, and synchronization). Each of these modules can be further characterized by smaller modules. For instance, the physical configuration of the

memory hierarchy is composed of global memory modules, cluster memory modules, local memory modules, and cache memory modules. Features of the modules can be refined by further decomposition. Different ways of organizing the machine features have different advantages and tradeoff. Our knowledge representation scheme allows the application to choose the desired way of organizing the knowledge according to the requirements of the application. Figure 5.3 shows two different ways to represent the hierarchical structures of the parallel computers based on different characterizing schemes. In figure 5.3(a) the organization is simpler but features are not as detailed as the ones shown in figure 5.3(b). For applications requiring minor optimization, the one in figure 5.3(a) may be enough, but for more complicated optimization, a finer representation such as the one in figure 5.3(b) would be better. As a matter of fact, the representation can be refined by adding more feature objects and relationship functions as the system implementation progresses. Thus, this refinement can be done incrementally without affecting the part of the software that has already been done, although the behavior of the system should be improved with more detailed machine knowledge. One good strategy in adding new machine features is to check if this added feature can distinguish between conflicting knowledge and help in making a better decision. On the other hand, new features may lead to discovery of new relationships or heuristics.

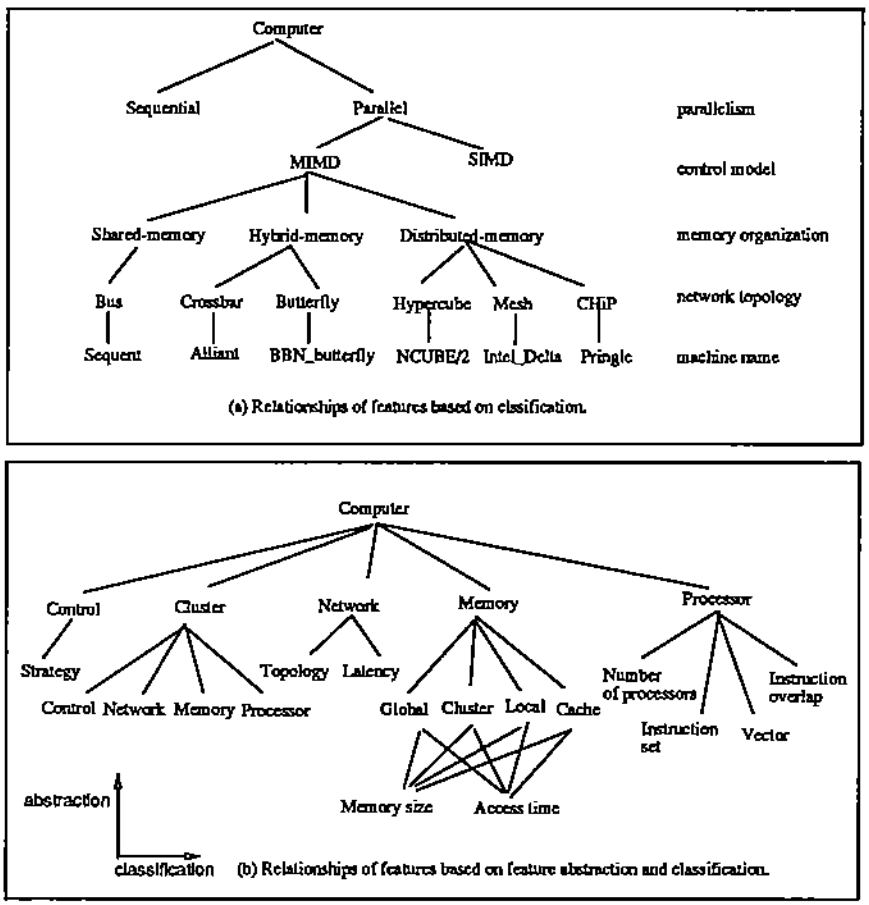

Figure 5.3. *Two highly simplified examples for characterizing parallel architectures.*

5.4.6.3. Various Abstraction Levels of Features

A key idea in automatic generation of high performance parallel programs is to express the knowledge of the machine and the program at an appropriate level of abstraction. Abstracted features of the machines range from high level concepts such as "shared-memory MIMD" or "distributed memory MIMD" or "topology of the interconnection network" to detailed properties such as "vector startup time" or "memory reference costs." The most appropriate abstraction level for the specification of the machine knowledge depends on the current state and goals of the compiler and the types of applications that utilize the parallelism of the machine.

Different types of applications will require different levels of abstraction to express the computation model of the application. In some cases, it is determined by the extent of optimization that the compiler is seeking. For example, to decide the patterns of data movement, only the topology of the communication network is required. But to get the optimal data movement, other information such as the dimension of the hypercube, the startup and unit costs for message transfers, network protocols, and optimal communication/computation ratio for the architecture are needed. Allowing different abstraction levels simplifies the implementation of parallel compilers, since different tasks of the compilers can deal with computational models at different abstraction levels that are suitable for the tasks. On the other hand, it complicates the implementation of the knowledge representation. Our program representation scheme allows dynamic abstraction of the knowledge by encoding the relationship between the knowledge at different abstraction levels. And the machine knowledge can be mapped to the appropriate abstraction level at the run time of the compiler.

5.4.6.4. Global Visibility Versus Knowledge Hiding

Data abstraction hides the internal data representation of an object from the uses of the object. This concept has been proven to be useful in conceptual abstraction of the objects and in the practice of modular programming. On the other hand, there are many cases where the ability to access the states of the objects is desirable. For example, the objects that define the relationships among objects are better visible globally. This is particularly useful in keeping the representation scheme flexible enough so that accommodating new architectures is easy. Also, a uniform structure for systematic access and support for representing heuristics also requires individual features to be accessible globally. Our representation scheme provides some mechanisms for knowledge to be encapsulated in objects but also globally visible.

5.5. Implementation of the Machine Knowledge Manipulation System

In this section, we provide some details of a prototype machine knowledge manipulation system that we implemented to support the parallel compiler we are constructing (see chapter 8). The knowledge manipulation system uses the object-oriented representation scheme we described above, and it supports interactive feature specification, update, query, and reasoning. It is designed for parallel compilers but may also be applied to other software systems require detail hardware knowledge.

5.5.1. A Machine Knowledge Manipulation System

The machine knowledge representation system consists of a machine feature database, an inference engine, an SQL relational database interpreter, and an interactive machine feature specification system.

The machine feature database contains three kinds of knowledge about machine features: knowledge of feature definitions, knowledge of feature usages, and knowledge of features of parallel computers. The inference engine can be used to compare and deduce features to help the specification and classification of the features. A subset of the SQL relational database language is implemented in Prolog to compare the features of the machines and help the knowledge expert to abstract machine features from heuristics; this provides a powerful mechanism for the manipulation of machine knowledge. The interactive machine feature specification system provides the man-machine interface for interactive specification and manipulation of the features of parallel computers. The interactive machine feature specification system is interfaced to both the SQL database server and the inference engine so that the user can query or analyze features of the machines.

The machine knowledge manipulation system is implemented in Prolog. The system is menu-driven and the user is allowed to pick a fact or predicate known to the system from the menu or to specify a new fact interactively. Details of the procedures for machine feature installation are given in the next section.

To effectively utilize parallel computers, a program optimization system needs to have enough knowledge in two areas: the hardware features of the machine and the heuristics of using the machine. The machine knowledge manipulation system can assist the parallel compiler writer in manipulating the parallel machine knowledge by providing the following functionalities: specifying of new machine features to the system, specifying of machine features for a new machine, finding relations of a feature with other features, comparing features of different machines, and supporting the reasoning for intelligent compilers. The last three abilities are especially important when analyzing the machine features to construct the system or collecting new heuristics to enhance the capability of the system.

The process of installing new knowledge includes identifying, translating and representing new machine features and heuristics. Human interaction is needed for this process, but systematic assistance from the

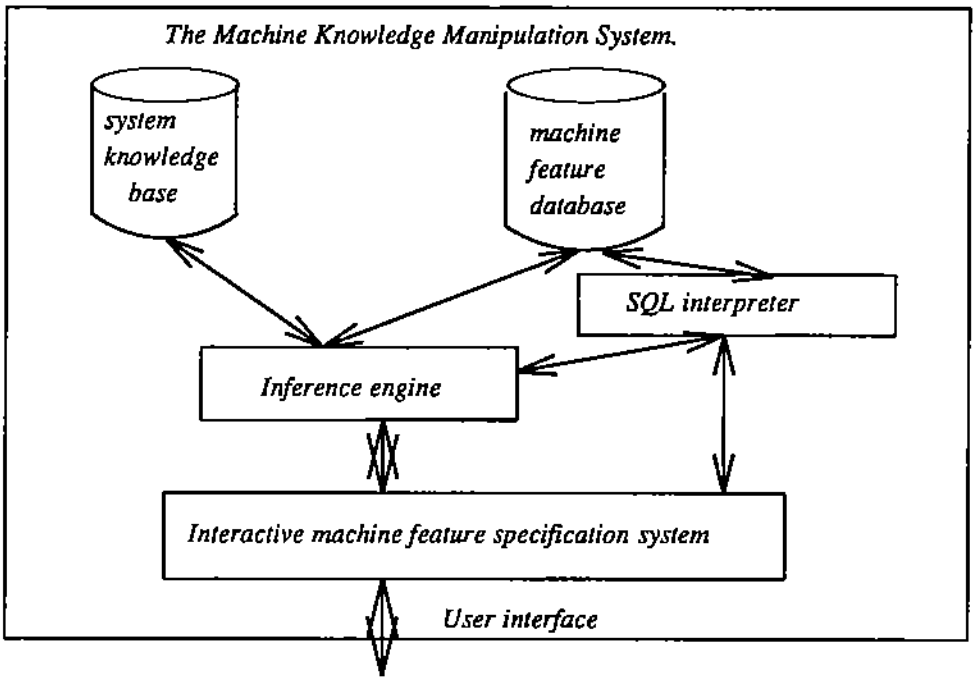

Figure 5.4. *The machine knowledge manipulation system.*

system can reduce the complexity of the task significantly.

5.5.2. Machine Feature Abstraction and Installation

The process of collecting machine features is illustrated in figure 5.5. The figure shows four components in the process: installing features for the machine, installing heuristics for the machine, installing new machine features to the system, and installing new heuristics to the system. These procedures are interrelated and can be carried out interactively.

5.5.2.1. Interactive Machine Feature Specification

New machines can be added to the system interactively with a user interface that allows the user to specify the features one by one. The system composes queries to ask for the features of the new machine based on its knowledge in its database and the machine features specified so far. A top-down approach based on the hierarchical structure of the known machine features is adopted and the query session begins with high level specification of the machine, such as computational modes, distributed or shared memory, and gradually gets to the details of the machine features. The user does not need to know the structure of the machine features. Based on the pre-conditions and the organization of the features, the system is intelligent enough to ask only for the related features. There are also commands to allow the user to input features that the system did not ask for or does not know at all.

This interactive machine feature specification contains three kinds of activities:

1. The system asks the system programmer values of the related features of the machine.
2. The programmer specifies the features that the machine does not know about.
3. If the feature specified by the programmer is new to the system, the user also needs to specify relationships between the feature and other features, possibly with help from the machine knowledge manipulation system. The system will also help the user in specifying the value of the new feature for machines already in the database of the system but whose corresponding feature values have not been specified yet.

To specify a feature that the system already knows (has feature object definition for the feature), the system prompts the user for the value of the feature with a menu dynamically constructed at run time. The user can also input the values of the features through the keyboard.

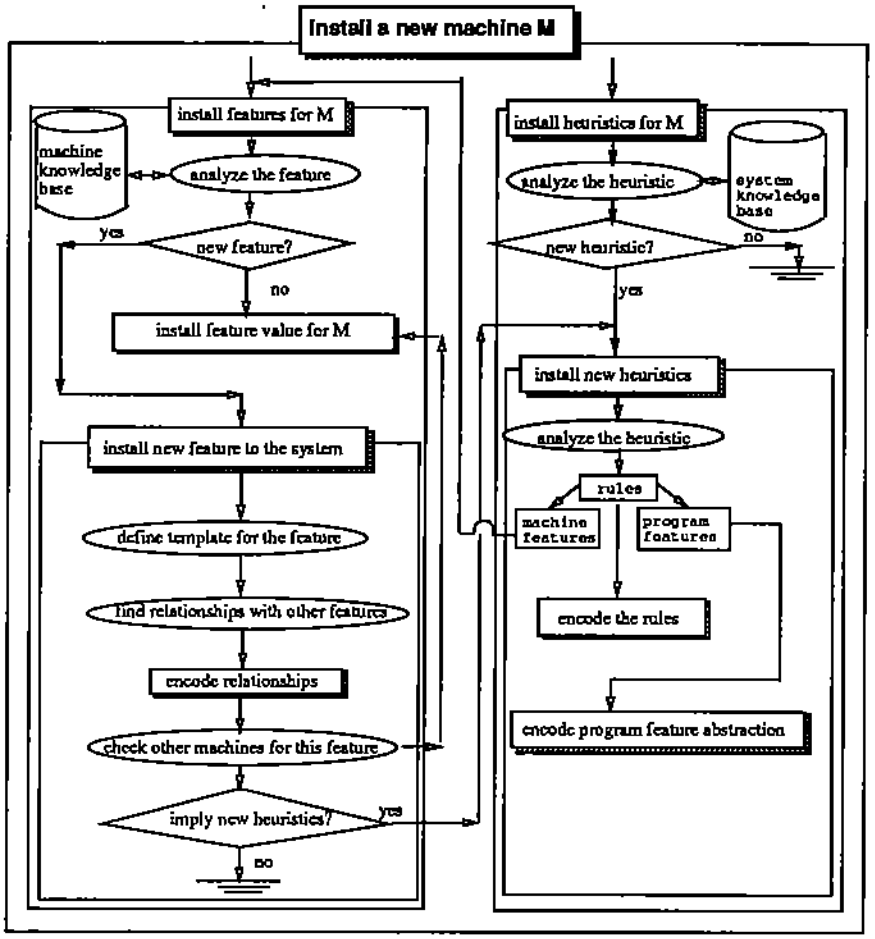

Figure 5.5. *Machine feature specification and installation.*

A new feature is added to the system by defining the template of the new feature (by defining the feature object) first. This process is illustrated by the example as shown in figure 5.7.

After the new feature object is defined, the next step is to find relationships between the new object and other features. This process is normally non-trivial and the reasoning ability of the system may provide the user with help. After the user specifies some basic relations, the system tries to help by finding all features that are related to these features and providing this information to the user.

Finally, after the new feature is installed, an attempt is made to relate this new feature to the parallel computers already known to the system. In other words, the system will try to enhance its data base by figuring out whether other machines in the database have this feature, and if so what the values of the features are for these machines. The user will have to decide whether the new feature can be applied to particular target machines in the knowledge base of the system, but the system will be able to rule out machines that cannot have the feature based on the relationship specified by the user.

After the machine features are installed, the heuristics of using the machine can be installed. One problem is to figure out what machine features are involved in a given heuristic. The machine feature comparison and deduction provided by the machine feature manipulation system is very useful. To specify a heuristic, the user uses menus to specify the preconditions and the actions of the rules. The menu can lead users through the hierarchy of machine features from the top down and the knowledge base keeps a list of abstractions of the program features so that the user can relate machine features and the program features to the heuristics. The structure of the hierarchy and the computational models helps the user in analyzing the relationship between the features and the heuristics. After the related features are picked, the system uses information in the feature objects to generate the preconditions and actions for the heuristics and translates them into rules. If the heuristic involves features that are not in the knowledge base, then the new features need to be installed.

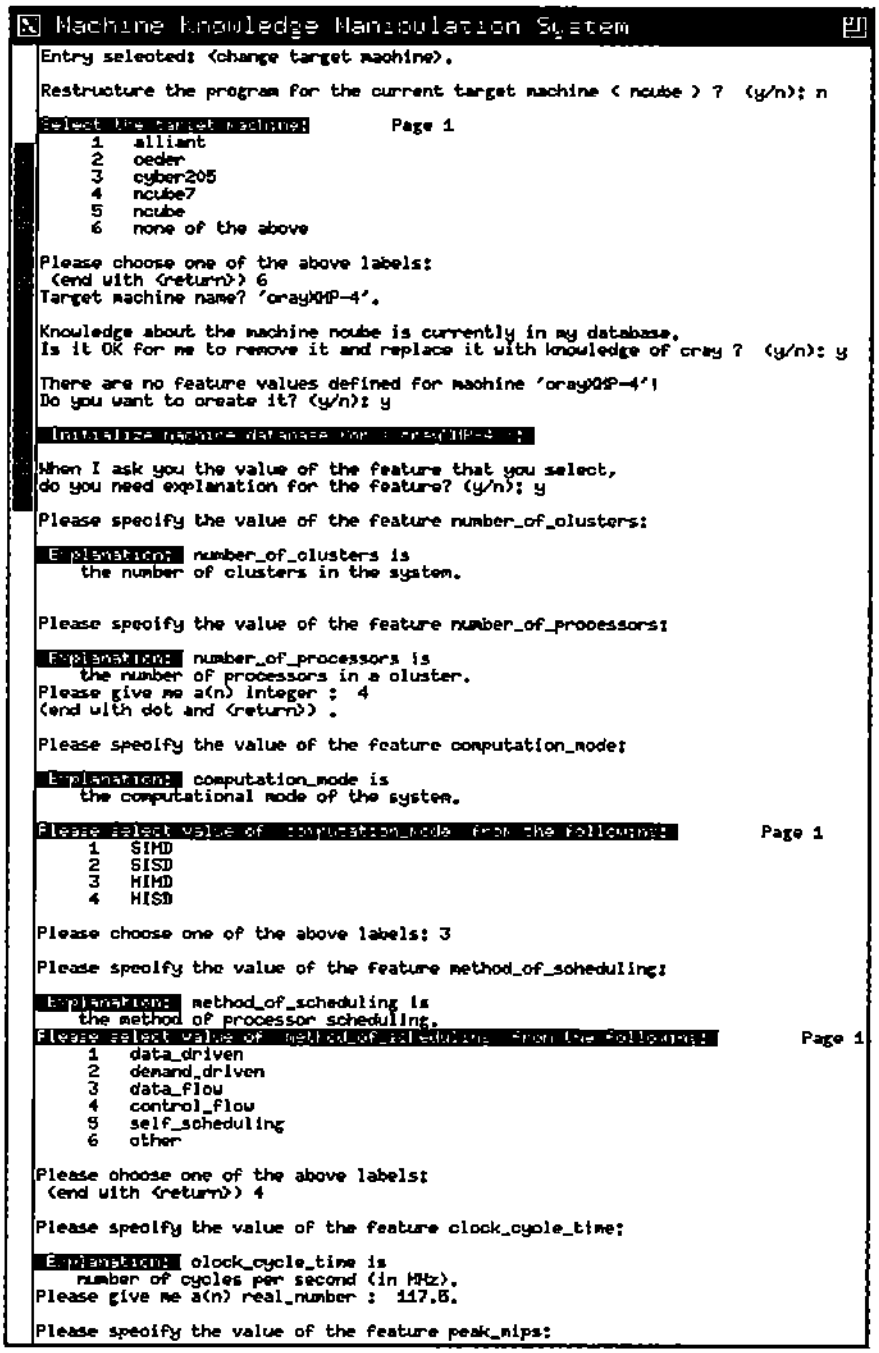

Figure 5.6. *An interactive session to specify features of a target machine.*

As we noted above, getting a complete description of the machine features is difficult but is usually unnecessary. Using a partial list of the machine features to represent the machine adds two requirements in manipulation of the machine knowledge:

1. Knowing what features are relevant to the system and
2. Knowing the relationship of the new knowledge with the existing knowledge of the system.

One simple methodology to decide what machine features to represent is based on the heuristics of the system. When installing the heuristics, users represent the heuristics and methodologies of utilizing the machine parallelism as a function of the machine features and program properties. After the user finishes the

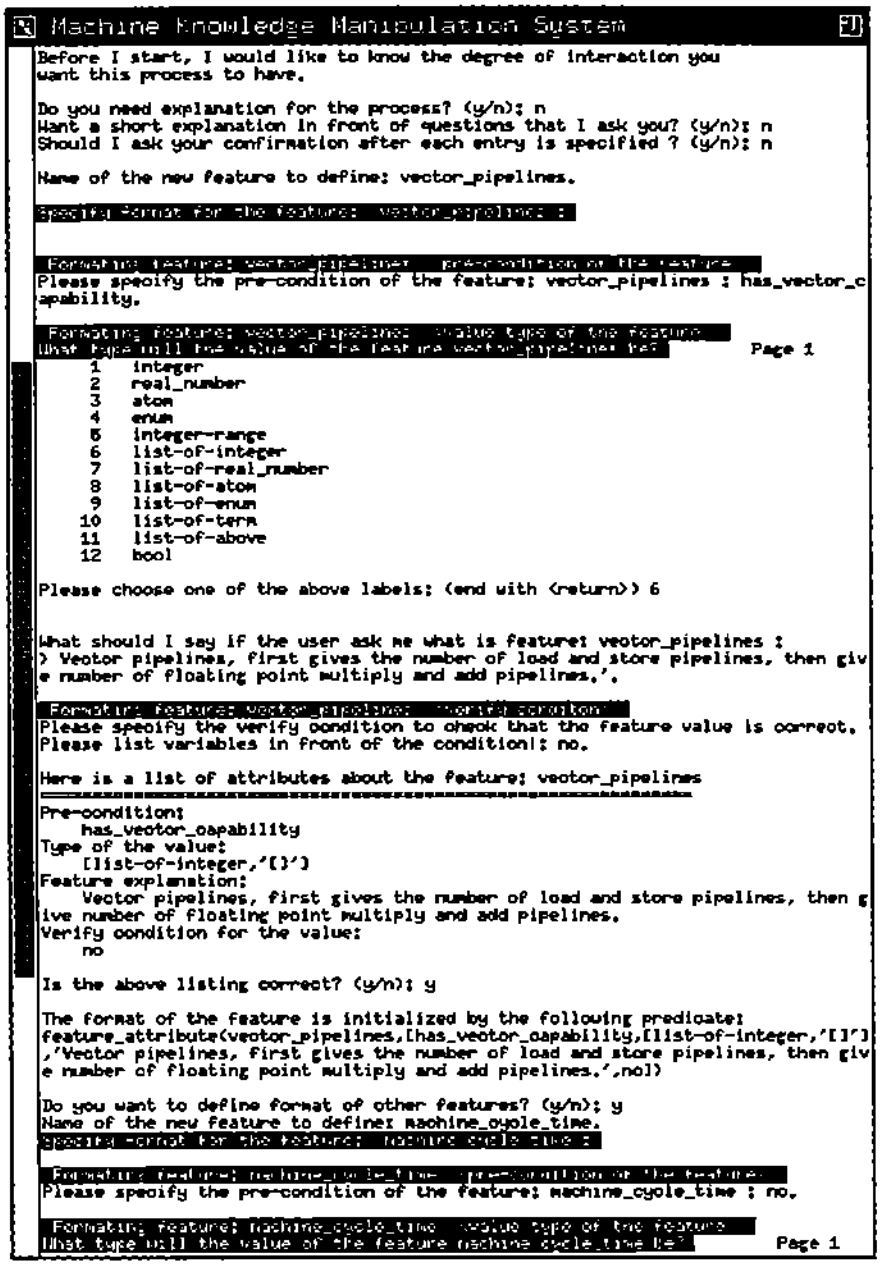

Figure 5.7. *An interactive session to specify template of a feature.*

specification of the heuristics, the system collects the list of machine features used and compares the list with the list of features that it already knows. In this way, new features can be discovered and installed systematically.

The procedures outlined above rely on two things: human interaction and the reasoning ability of the knowledge manipulation system to help the human expert sort out the complex relationship between the heuristics and the machine features. Systematic knowledge manipulation can significantly reduce the complexity of the machine knowledge abstraction and installation process. From our point of view, this is one of the basic requirements for all retargetable software systems.

5.5.3. Feature Deduction and Comparison

Heuristics are knowledge without a theoretical background. In order to generalize the heuristic to other parallel computers, heuristics need to be analyzed to determine the fundamental elements behind the heuristics. In this way, a new target machine can utilize the heuristic if the machine possesses all the features involved in the heuristic.

The ability to analyze relationships between machine features is needed because not all machine features are represented at the same level. Some features may be derived from other features and some may be obscured by other features. Similarly, when trying to abstract the features of a machine or distill effective features from a new heuristic, it is often necessary to compare features of the machines. The machine knowledge representation scheme we propose supports both operations plus other reasoning mechanisms. A knowledge base that features a simplified SQL database language plus inference mechanism is implemented to support the task of analyzing machine features and heuristics.

For example, it is possible to collect all machines that have feature F or find all features that can be derived by a set of features with the simple reasoning mechanism we described above.

Feature deduction is supported by the relation functions which are part of the object-oriented representations. Feature comparison of different machines is available by the implementation of a relational database. For example, we can find all common features or different sets of features of two machines with a relational database command: "find all common features of A and B" or "find all features A subst B."

```
select rule from machine
where machine.memory.cache_size > 0
      and machine.processor.special_intstructions.block_transfer = P
      and nonvar(P)
```

Figure 5.8. *A sample query to retrieve all rules that the system know about block–transfer for machines with caches.*

5.5.4. Specializing System Knowledge

The advantage of having a general purpose machine knowledge manipulation system is that knowledge can be accumulated and shared among different architectures. The price paid is that when reasoning is performed, the performance suffers because of the added tests for checking the applicability of the heuristics at the compile time. We use a methodology called *knowledge caching* to improve the performance of the inference system. The method works as follows: As the compiler is being constructed, maintained and enhanced, the system encodes the knowledge with the multiple target paradigm. By the time the compiler is ready to be distributed for a particular machine, the vendor may choose to specialize the compiler for a particular target machine by validating or invalidating rules in the knowledge base. This is done by pre-evaluating the rules based on the features of the target machine. All conditions that are implied by the static features are eliminated from the rules, and the resulting simplified rules are "cached" in the specialized compiler. Also, rules that are disqualified by the static features are deleted from the knowledge base of the new compiler. This approach has the advantage of code re-use and knowledge sharing but does not suffer loss in system efficiency.

The method is called knowledge caching because it also provides a runtime mechanism to optimize feature access. The same procedure can be used at run time to further eliminate redundant conditions or invalid rules based on the dynamic features. The most recently accessed or generated objects are kept in the system memory so that subsequent access will be much cheaper. On the other hand, if other information is needed, they can be loaded when the system detects that it lacks the information (this is supported by an indexing mechanism supported by the underlying expert system shell described chapter 7). The resulting parallel compiler is specialized for the target machine with the features of the machine implicitly built into the rule base of the system. The rules for the program transformation can be linked with the machine features by attributes of the features. Therefore, after the features of the machine are specified, a personalized knowledge base can also be built for the particular architecture and thus improve both space utilization and execution efficiency.

5.6. Other Applications of the Representation Scheme

The machine knowledge manipulation system we described above can be applied to many other software systems that need details of the target machines. Two such examples are given here.

5.6.1. Distributed Computing Environments

A distributed computing environment is a system that contains a set of loosely connected computers. Although not every distributed system needs detailed machine information of the computers in the system, some applications do require the system to have low level knowledge of its members. An interesting example of this is as follows: in a network that contains a wide spectrum of architectures (for example, a network of workstations, graphic workstations, main frames, and supercomputers), and applications. Suppose the goal of the operating system is to assign a task to a machine that is most appropriate (the objective may be adaptive) for the application; the system requires a clean understanding of the capabilities of each machine in the system and some information about the applications to make a smart decision. Our machine knowledge manipulation system allows the system to possess and manipulate knowledge of many computer systems and support the match of machines and applications. Thus, it is possible to build a smart task scheduler for distributed, heterogeneous, computing environments based on the machine knowledge manipulation system.

5.6.2. Flexible Simulation Systems for Parallel Architectures

A product design cycle consists of requirement, specification, design, prototyping, testing, and modification. The design of new computer architectures usually goes through this cycle many times before the final product emerges. At each iteration of the design cycle, requirement, specification, and design of the machine may be changed because of technical difficulties, marketing considerations, and other unexpected problems. Any changes can affect the subsequent phases and complicate the designing problem. Building a hardware prototype is very time-consuming and expensive. In contrast, an electronic prototype can shorten the design-testing cycles and decrease the product-developing lead time. The machine knowledge manipulation system we discuss here provides a foundation for building software simulation systems for parallel computers. When coupled with the performance evaluation system discussed in chapter 6, a very flexible general purpose parallel architecture simulation system can be built. Under this model, revising the architecture design is relatively easy since the machine knowledge manipulation system provides easy modification of the machine feature entries. Furthermore, when integrated with the program transformation system, the resulting system has two significant advantages:

- The domain that the architecture is targeting can be used to test and evaluate the design before a hardware prototyping system is built. Problems in design can be discovered at the early phase of the developing cycle.
- The experience accumulated from other parallel computers can be applied to the new machine. Thus, a great deal of heuristics for using the machine exist even before the machine is actually built.

5.7. Conclusion

In this chapter, we have presented a machine knowledge representation scheme for parallel computers that supports reasoning. Intelligent reasoning is possible because decisions can be made from analysis of machine features. The machine knowledge manipulation system forms the basis of a parallel programming environment we are implementing which can restructure the program structure intelligently.

The knowledge representation scheme we propose has the following significant features:

1. Knowledge sharing. Separating the machine knowledge from the system heuristics allows heuristics to be shared among different parallel computers.
2. Object-oriented representation. The object-oriented representation scheme allows modular and elegant knowledge representation and ease of manipulation. With abstraction and classification operations, the parallel machines can be abstracted into different levels of computational models.
3. Tolerance. Incomplete machine specification as well as incomplete system knowledge is allowed in this representation scheme. When a feature is not present, the system simply assumes that the target machine does not have it. Even though the performance may suffer, the more detailed knowledge about a machine the system contains, the better the approximation of the computational model is to the real machine. However, the system works even with incomplete knowledge of the machine so that

knowledge about a new machine can be incorporated incrementally.

4. Flexibility in the organization of the knowledge. The machine knowledge can be organized on different criteria. For example, at the higher level, the machine knowledge can be classified on the levels of parallelism (the multiprogramming level, multiprocess level, inter instruction level, and micro instruction level). At the lower level, the machine knowledge can be grouped on the physical organization of the system (processing units, memory hierarchy, communication network/bus, clusters of processors, and control unit).
5. Test beds for complex systems. The proposed representation scheme allows easy construction of test beds for complex software or hardware systems. Features and relation functions can be added or removed to test the consequence of the action. This property of the system can also be used to evaluate or study the effectiveness of the features, relation functions or heuristics.

Although we are targeting the methodologies to the parallel compilers, the methodologies can be applied to any software systems that require detailed knowledge of the underlying architectures, such as parallel simulation systems, distributed operating systems, and runtime environments.

The entire machine knowledge manipulation system and the SQL database interpreter are implemented in Prolog. The feature objects are stored as facts in the internal database of the Prolog interpreter. The user interface is still clumsy but is expected to improve greatly when the X Window interface is constructed.

CHAPTER 6
PERFORMANCE PREDICTION AS A BASIS FOR INTELLIGENT PROGRAM OPTIMIZATION

6.1. Introduction

Optimizing parallel compilers use heuristics and program transformation techniques to modify the program structure to improve the parallelism of the user programs. The ability to accurately estimate the performance of a program on a target machine and assess the improvement in performance or other aspects that a transformation can have for the program under consideration is vital in the decision-making process of parallel compilers. This process, call the *performance prediction*, involves evaluating the match between the characteristics of a program and the constraints of an architecture and then using this match to estimate the performance of the program.

The most important criterion for a performance prediction module in an intelligent parallel compiler to meet is that it needs to be *inexpensive, accurate* and *flexible*. It has to be inexpensive because the compiler needs to use it repeatedly for evaluating the merits of the applicable transformations during the program restructuring process. It has to be accurate because proper decisions of the compiler rely on a good estimation of the performance. It has to be flexible because the program optimization process in an intelligent parallel compiler uses different objectives at different stages of the process and for different architectures. A mechanism to fine-tune the performance prediction model to suit different objectives is needed so that the system can adjust itself to face different challenges in the decision-making.

In this chapter, we describe an analytical performance prediction model for estimating the performance of programs on different classes of parallel architectures with high accuracy and efficiency. We define a template for implementing this analytical model so that it can be integrated into the decision-making process of a parallel compiler. Many performance factors are considered and listed in section 6.4. One distinct feature of our model is that it is extremely flexible: different performance prediction functions can be incorporated for different purposes, and different amounts of resources can be utilized. The performance prediction model utilizes some heuristics to handle unstructured programs, loops with unknown bounds, conditional statements, and can minimize run-time tests for these cases. This makes it an ideal candidate for intelligent parallel compilers.

6.2. Performance Prediction Models

The performance of a program on an architecture can be estimated with either the *simulation* or the *characterization* model. The simulation model estimates the performance of the program by simulating or profiling the execution of the program or interpreting the execution-time on a computational model of the architecture. The accuracy of the prediction depends on how faithful the computational model is in simulating the actual hardware and whether the performance of the program is sensitive to the input of the program. The characterization model characterizes the performance of the machine by certain aspects of the machine. The accuracy of the characterization model depends on how representative the selected aspects are. This model utilizes a set of easy-to-compute functions called *performance factors*. Each *performance factor* represents a particular aspect of the performance of programs on a target machine and is quantified by a corresponding function called the *performance evaluation function*. Each performance evaluation function is a function of the program fragment, P, and the target machine, *Machine*. The estimated performance, EP, is a function that combines the results of the selected performance evaluation functions E^i.

$$EP(P, Machine) = \sum_{i=1}^{m} g^i(E^i(P, Machine)). \qquad (6.1)$$

where E_i, $1 \le i \le m$ are m performance evaluation functions, and g^i is an adjustment function for E^i. When $g^i(E) = Weight^i * E$ for all i, then EP is a weighted linear combination of the evaluation functions E^i. That is:

$$EP(P, Machine) = \sum_{i=1}^{m} Weight^i * E^i(P, Machine). \quad (6.2)$$

Variations of the evaluation function model that use a certain formula to quantify the performance of programs have been applied to many other parallel compilers. However, most of these applications use fixed equations to compute the with little consideration to machine variance and objectives of the compiler. This has the following problems:

1. *Not suitable for a wide variety of architectures.* Most models do not take the architecture variations into consideration. Parafrase [Husm86] uses several different parameters for computing program execution time for seven different virtual shared-memory machines, but the number of parameters that are considered are too few to be accurate for real parallel architectures.
2. *Inflexible for different needs at different stages of parallel compiling.* A more critical part of the program requires more accurate but expensive performance estimations. Existing performance prediction models are inflexible to accommodate different objectives of the program optimization.
3. *Cannot deal with programs that contain variables in control flows.* Run-time tests are essential for programs that have dynamic behaviors. The existing performance prediction model does not have provision to decide the minimum run-time tests that are needed. This can lead to excessive run-time overheads.

In the next section, we propose a highly flexible performance prediction framework that can be integrated into intelligent parallel compilers to solve the above problems. The framework, which is called *refined combined characteristic* model, is augmented with a knowledge base which can dynamically selects evaluation functions to apply based on features of the machine and objectives of the program optimization. Different weights can be given to the evaluation functions dynamically so that their effects can be adjusted. Inexpensive performance evaluation functions are combined into a single-valued real function as an indication of the performance of the program on a parallel architecture and to compare representations of semantically equivalent programs.

6.3. A Framework for Performance Prediction

Based on the characterization model, a framework for predicting performance of programs on different parallel compilers can be defined. This framework consists of a rich set of performance evaluation functions, and a control to choose the evaluation functions and decide how to combine the results.

The control of the performance estimation can be divided into four modules: the prediction-setup module, the prediction-construction module, the prediction-update module, and the prediction-refining module. Each of these modules can be adjusted by rules in the knowledge base based on the objectives of the compiler. A list of evaluation functions is discussed in the next section. We will examine the four modules first.

The prediction-setup module is invoked by other modules to prepare for prediction estimation. Its tasks include selecting the set of performance evaluation functions to use, setting architectural-dependent constants for the selected evaluation functions based on the machine features, setting weights for the selected evaluation functions based on objectives of the optimization, and choosing the method for accumulating and combining the results of the evaluation functions.

The prediction-construction module estimates the performance of the program by applying the set of evaluation functions to each node in the program dependence graph according to a depth first search order based on the control dependence (with back edges being ignored). The process of performance construction can be decomposed into the following steps. First, it invokes the prediction-setup module to set up the constants. Then, the evaluation functions are applied recursively on elements of compound statements and the results are aggregated by an accumulation procedure. Third, the prediction-combining procedure is applied to integrate the results of the evaluation functions.

A simple example of the combining procedure is the weighted linear combination as described in equation (1) in section 6.4; more complicated combination functions can be supplied by the user. Also, depending on the setup, the performance results can be combined while they are computed at each node or they can be computed individually and combined at the task level.

The performance-construction process can be summarized in the following procedure:

```
PerformanceConstruction(Node, PDG, ListOfEvalFunctions, Results)
Begin
  if (Node is a compound statement) then
    for each Child of Node in PDG do
        PerformanceConstruction(Child, PDG, ListOfEvalFunctions, Result0);
    endfor
    Accumulate(ListOfEvalFunctions, Result0, Result1);
  else
      Evaluation(Node, ListOfEvalFunctions, Result1);
  endif
  if (combine_at_node or Node is a task) then
      Combine(ListOfEvalFunctions, Result1, Result);
  else
      Result = Result1;
  endif
end
```

Figure 6.1. *Performance construction procedure.*

The prediction-update module is applied during the course of the program optimization process. The evaluation functions involved in this process estimate the *changes* that a transformation might have on the performance of the program and adjust the prediction accordingly. The performance evaluation functions and the method to combine the result used in this module do not need to be the same as those used in the performance-construction module. Only the region of the program that will be affected by the transformation needs to be updated. Also, the overall performance prediction can be adjusted incrementally.

The prediction-refining module can be engaged optionally during the program optimization process to refine the estimation by using a set of more detailed evaluation functions. It can either compute the estimation from scratch using a set of more sophisticated evaluation functions or it can refine the original estimate by updating certain aspects of the estimation.

6.3.1. Dynamic Performance Prediction and Run-time Tests

There are programs for which the static analysis of the compiler may fail to predict the control flows of the programs. These cases are normally programs that have variables or indirect references in the control structures (such as loop bounds or conditions), or conditional statements for which probabilities of the branches cannot be estimated. These cases present challenges to the performance prediction.

If the unknown variable involved in the estimation is a loop index, the problem can be easily solved. For example, for the following nested loops, the loop bound of the inner loop depends on the outer loop, but can be easily computed.

```
for i in 1.. N loop
      for j in i+1 .. k * i loop
            .....
      end for;
end for;
```

If the evaluation of loop instant j of the inner loop j is $f()$, then the evaluation function for loop j is $\sum_{j=i+1}^{k*i} f() = (k-1) * i * f()$. And the evaluation function for loop i is $\sum_{i=1}^{N} (k-1) * i * f() = \frac{N * (N+1)}{2} * k * f()$.

For programs where unknown variables appear in the loop bounds or conditions in the conditional statements, the performance prediction model we described here can be applied by utilizing the inference capability of the system. For each performance evaluation function there is a list of parameters that are used to compute the evaluation. The inference engine evaluates the evaluation function by unifying the variables with their

values first. If variables remain undefined after the unification stage (for example, if there are variables in loop bounds), the evaluation function will be evaluated as a function of the undefined variables by a common compiler optimization technique called *constant folding* [AhSeUl86].

Operations (such as adds, multiplies, comparisons) can be performed on results of evaluation functions (including those with non-instantiated variables) to merge different evaluations or use them to compute other evaluation functions.

For variables that do not depend on the input, the values of the variables or the bounds of the variables can be estimated by an AI technique called *constraint propagation*. This can normally solve the problem or limit the range of the performance estimation.

For variables that depend on the input data, there is not much a compiler can do to estimate the values of the variables. It can then either query the user for possible values or generate some conditions on the performance based on the uninstantiated variables in the performance function. These conditions can be used to generate run-time tests when the control decisions need to be decided at the run-time.

6.4. Performance Evaluation Functions

In this section, we first discuss how performance evaluations can be defined and combined, then give a survey of some useful performance prediction functions.

A performance evaluation function is a quantification of the quality of an aspect of the match between the program and the machine. Typically, the definition of an evaluation function has two parts: evaluation of a non-compound statement and the accumulation of the results of applying the function to components of a compound statement. For example, the equation (6.3) defined below is used by several of the evaluation functions defined in section 6.4.1 to aggregate results of applying an evaluation function recursively on children of a compound statement.

$$f(S) = \begin{cases} \sum_{i=1}^{n} f(S_i) & \text{if } S = \{S_1, S_2, \cdots, S_n\} \\ \max_{i=1}^{n} f(S_i) & \text{if } S = \{S_1 \mid S_2 \mid \cdots \mid S_n\} \\ \sum_{i=l}^{u} f(S(i)) & \text{if } S = \text{for } (i = l \,..\, u)\ S(i)\ \textit{endfor} \\ \max_{i=l}^{u} f(S(i)) & \text{if } S = \forall\ (i = 1 \,..\, u)\ S(i)\ \textit{endfor} \\ p(S_T)*f(S_T) + p(S_F)*f(S_F) & \text{if } S = \text{if } (\textit{cond})\ S_T\ \textit{else}\ S_F\ \textit{endif} \\ f(S_{fb}) & \text{if } S \textit{ is a function and } S_{fb} \textit{ is its body} \\ F(S) & \textit{otherwise} \quad \ldots\ldots\ (*) \end{cases} \tag{6.3}$$

where $p(S)$ stands for the probability that the condition statement will branch to statement S and $S(i)$ stands for the i–th instance of the loop statement S. Also, $\{S_1 \mid S_2 \mid \cdots \mid S_n\}$ denotes that the statements S_i are to be executed concurrently.

Evaluation functions can be defined by changing the definition of $F(S)$ in the line marked with (*) in equation (6.1) into some specific functions.

6.4.1. Examples of Performance Factors and Their Evaluation Functions

The evaluation functions defined by the performance factors map the analytical match of a program and a machine into numerical values. Below we list some sample performance factors and their corresponding evaluation functions. Each of these performance factors represents a particular aspect of the performance of the program. In the next section, we will discuss methodologies for integrating these factors into a framework for predicting program performance on target machines.

- *Statement count:*

The statement count characterizes the algorithm by counting the number of statements in each of the control threads. The execution time of the program may be estimated by multiplying by a constant called the

average_statement_cost with the statement count. The statement count can be computed by replacing $F(S)$ in equation (1) by

$F(S) = 1$ if S is not a compound statement

• *Operation count:*

A more accurate estimation, called operation counts, can be computed by counting the number of operations instead of just the statements in the control threads. An evaluation function for computing the operation count $f()$ for a program fragment S can be obtained by replacing the line marked with (*) in equation (1) with the following definitions:

$$F(S) = \begin{cases} f(expr) + 1 & \text{if } S = operand\ op\ expr \\ \alpha^v + \beta^v * n & \text{if } S \text{ is a vector operation of length } n \\ 0 & otherwise \end{cases}$$

where α^v and β^v are constants representing the vector startup time and unit time using the cost of a floating point operation as units.

The estimated program execution time can be obtained by multiplying the operation count with a constant called *average_operation_cost*. For RISC processors, this constant can be computed easily at the assembly instruction level since most manufacturers have studied and made assumptions about the distribution of operations in the target applications. On the other hand, for CISC processors or for operations at the language level, the variation in the execution time of operations can be quite large, as is the discrepancy between the estimated operation cost and the actual cost. An improvement to this approach is to classify the operations into groups of different costs and use different average-time estimations for each operation group. For example, integer operations often take less time than floating point operations, additions usually take less time than divisions, and array references usually take several integer operations to compute their addresses. A even more detailed estimation can be done by using the actual cost of each of the operations. For example, the machine knowledge base may record the costs of floating-point multiply, integer addition, array address calculation, startup and element time for *vector add* operations, and startup time and element time for triadic operations, etc. Note that the costs of some operations (especially floating point operations) on certain machines take variable time depending on the operands, so estimations of the average costs of these operations are still needed.

• *Number of memory loads and the number of memory stores.*

The number of memory loads and stores are two special types of operation counts that we would discuss separately. The unit cost for memory load and store operations (C^{load} and C^{store}) are usually constant. So the cost of memory access may be estimated by counting the number of memory load and store operations. For machines with a multiple level of memory hierarchy, the number of global memory loads, global memory stores, cluster memory loads, cluster memory stores, local memory loads, and local memory stores should be counted separately since their costs are usually different.

For machines that support block-accesses, the memory load and store costs for referencing a data block of size n, T^{load}_{block} and T^{store}_{block}, can be computed by:

$$T^{load^t}_{block}(n) = C^{load^t}_{start} + C^{load^t}_{unit} * n$$

$$T^{store^t}_{block}(n) = C^{store^t}_{start} + C^{store^t}_{unit} * n$$

where the superscript t in above formula is either g, c, *or* l, which stands for global, cluster, and local memory, respectively. Also, $C^{load^t}_{start}$, $C^{load^t}_{unit}$, $C^{store^t}_{start}$, and, $C^{store^t}_{unit}$ are constants for the starting cost for block-load, unit cost for block-load, starting cost for block-store, and unit cost for each block-store, respectively.

• *Cache hit ratio.*

The cache hit ratio of an array is the ratio of the references to the elements of the array that are already in the cache. It represents the degree of the locality of the array in the program and can be used not only in deciding cache allocation problems for architectures that have cache, but also in utilizing block-access operations in machines that support them. Following [GaJaGa87], we define the image in the computation of an array x of dimension d, $Im(x)$ as:

$$Im(x) = \{w \in Z^d \mid x[w] \text{ is referenced in the computation } \}$$

where Z^d is the bounded integer interval space of the indices of x. Assuming that R is the total number of references to array x in the program, the cache hit ratio is defined as [GaJaGa87]:

$$hit(x) = \frac{R - |Im(x)|}{R}$$

For example, in the following program fragment

```
for i := 1 .. N loop
    for j := 1 .. M loop
        for k := 1 .. L loop
            a[i,j] := b[i,k] * c[k,j];
        end for
    end for
endfor
```

the number R, which is the total number of references to array c, is equal to $N * M * L$. Furthermore, $Im(c) = \{(k,j): 1 \le j \le M, 1 \le k \le L\}$ and $|Im(c)| = M * L$. So the cache hit ratio for the array c in the program fragment is:

$$hit^{L_4}(c) = \frac{N * M * L - M * L}{N * M * L} = \frac{N-1}{N}$$

On the other hand, if we focus on the loop j, then $R = M * L$ and $Im(c)$ is the same, so the cache hit ratio of array c in loop j becomes 0.

• *Data transmitting cost.*

The time it takes to transmit a set of data of size n over the network or bus can be computed by the following formula:

$$T^{dt}_{block}(n) = \alpha + \beta * n$$

where α and β are the constants for start-up cost (for hand-shaking, setting up routing connections, etc.) and unit-transmitting cost. For distributed computers such as the hypercube where data may need to be transmitted by several hops through several processors, these two constants α and β can be computed by the following formula:

$$\alpha(hops) = \alpha^0 + \alpha^1 * hops$$

$$\beta(hops) = \beta^0 + \beta^1 * hops$$

where *hops* is the number of hops along the path. More precisely, the constant α^0 for a distributed machine actually includes the time for a zero-length send and a zero length receive; the constant β^0 is the time for the sending task to access the data and the time for the receiving task to store the data. As an example, these constants for NCUBE/1 and nCUBE 2 are shown in the following table (table 6.1).

Table 6.1. *Constants for computing message transmitting cost on nCUBE.*

Time (in microseconds)						
MACHINE	α^0	α^1	β^0	β^1	*mach. cycle*	*fl. add*
NCUBE/1	*261*	*193*	*4.25*	*4.5*	*0.137*	*2.47-3.84*
nCUBE 2	*158.5*	*1.3*	*2.49*	*0.0*	*0.05*	*0.35*

Therefore, sending a one word message to a neighbor takes 461.75 microseconds on NCUBE/1 and 162.3 microseconds on nCUBE 2. And sending a 1000-word message across a 128 node cube takes 37362.0 microseconds on NCUBE/1 but 2657.6 microseconds on nCUBE 2.

• *Voided data-prefetch ratio.*

For architectures that support data-prefetch, most data accesses can be overlapped with the computation if they are not void by conditional or unconditional branch statements. Therefore, the cost of data

synchronization is a function of the ratio of prefetched data that are voided by the control statements. A simple heuristic to estimate this ratio is to count the number of statements, N^{stmt}, in the statement block following a branch, and assume that all but one of them are benefited by the data prefetching. This gives us a ratio which is $\frac{1}{N^{stmt}}$. And the data synchronization cost can be estimated as:

$$C^{ds}(S) = C^{da}(S) * p^{vdp}(S) = C^{da}(S) * \frac{1}{N^{stmt}}$$

As the formula shows, the longer a non-interrupted statement block is, the more data access can be overlapped with the computation.

• *Ratio of data access overhead.*

This function characterizes the ratio of the non-overlapped cost of data access over the total execution time. The difference between this ratio and the voided data-prefetch ratio is that the first ratio is the data access overhead over the total execution time of the statement, whereas the second ratio is over the total data access time of the data in the statement. This means that this factor is used in case the computation of data access time is to be avoided. Of course, the non-overlapped cost of data access depends on the features of the architecture (prefetching mechanism, data cache, etc.) and the program (density of the local/external memory references, distribution of the external references, etc.) and is hard to estimate without computing the data access cost. This evaluation function assumes that the cost of data access is proportional to the cost of computation which may not always be true. We can estimate this cost by analyzing a set of representative program structures and finding the ratio of data access that can be overlapped with the computation, then using these patterns as the basis for interpolating non-overlapping data access cost for general programs. With this ratio, the data synchronization cost, C^{ds} over a program region S, becomes:

$$C^{ds}(S) = C(S) * p^{ds}(S).$$

where $C(S)$ is the cost for computing in S, and $p^{ds}(S)$ is the ratio of the data access overhead in S.

• *Do-across delay.*

A good measure for data synchronization cost is the do-across delays. Do-across delay is the artificial idle-time in the shared-memory environment that the compiler inserts into a do-across loop to minimize the load (the hot-spot effect) that a busy-waiting processor induced on the memory, bus, or network in checking the global synchronization data. This delay represents the expected data synchronization cost of the loop instance and is the minimum delay that satisfies all control (condition and unconditional branching statements) and data dependencies in the parallel loops. The do-across delay of a parallel loop is computed by finding the maximum static time interval between the pairs of statements involved in lexically backward data dependencies in the loop and then spreading the delays into loop instances [Cytron84]. Given the do-across delay d_{L_i} for loop L_i, assuming that the loop has N loop instances, the estimated execution time for distributing the do-across loop over p processors is given in the following formula:

$$T = \begin{cases} (N-1)*d_{L_i} + T_{L_i} & \text{if } T_{L_i} \le p * d_{L_i} \\ \left\lfloor \frac{(N-1)}{p} + 1 \right\rfloor * T_{L_i} + mod(N-1, p) * d_{L_i} & otherwise \end{cases}$$

• *Do-across parallelism degree.*

Do-across parallelism degree [Cytron84] is the reciprocal of the ratio of the do-across delay over the actual execution time of the loop body. Assume that T_{L_i} is the execution time of the body of loop L_i and d_{L_i} is the do-across delay for loop L_i, then the do-across parallelism degree of the loop can be computed as:

$$Para^{doacross} = 1 - \frac{d_{L_i}}{T_{L_i}}$$

It follows that given the parallelism percentage, the do-across delay can be computed by the following formula:

$$d_{L_i} = (1 - Para^{doacross}) * T_{L_i}$$

Doacross parallelism degree is an indication of the parallelism presented in the do-across loop.

• *Number of data synchronization points.*

Synchronization adds overhead and may cause processors to idle but is needed to enforce data dependencies that cross the task boundaries (to be more precise, for those dependencies that cross processor boundaries). To estimate the effect of synchronization on the proposed task scheduling, the number of cross processor data dependencies can be used. This number can be collected by simply counting the cross-task data dependencies and eliminating those among the tasks that are assigned to the same processor.

• *Uniformness of the execution time.*

Let $\{P_i \mid i = 1 .. n\}$ be a set of n tasks, $T(P_i)$ the estimated execution time of the tasks, T^{max} be the maximum of the estimated execution time, and T^{mean} be the mean of the estimated execution time. The uniformness of the execution time is defined to be:

$$Unif = 1 - \frac{\sum_{i=1}^{n} |T(P_i) - T^{mean}|}{n * T^{max}}$$

For tasks that have more uniform execution time, load balance is easier to achieve and static scheduling can be used. On the other hand, for tasks whose execution times vary, dynamic scheduling may be better.

• *Hot-spot percentage.*

A hot-spot is a module in the multi-stage blocking network that has sufficient concentration of the traffic. The non-uniform network traffic of hot-spots can produce effects (called tree saturation) that severely degrade *all* network traffic [PfNo85]. Although message combining is an effective technique for solving this problem when the sources of hot-spots are global shared locks, the hardware needed for supporting the combining is quite expensive (Pfister and Norton [PfNo85] estimated that the combined network increases the size and cost of the switch in a factor of 6 to 32).

Let the number of processors be p, and the number of network packets emitted per processor per switch cycle be r ($0 \le r \le 1$), the *hot-spot percentage*, ρ [PfNo85], is defined to be the fraction of the data references directed at the hot-spot (i.e. each processor emits packets directed to the hot-spot at a total rate of $r*h$). Then the effective data rate into the hot module is $r(1-h) + rhp$. And the asymptotic limit of the total communication bandwidth available is:

$$B = \frac{p}{1 + h(p-1)}$$

This function gives a limit on the available speedup for a given number of processors. The function was originally derived in [PfNo85] for shared memory-modules on multi-stage networks, but can be generalized to more general distributed systems. This limit imposed by the hot-spot degradation is very significant. For example, for a 1000-processor system, a hot-spot percentage of 0.125% can limit the potential speedup to 50%. The hot-spot percentage can be estimated by analyzing patterns of the data dependence graph. The hot-spot percentage can be used to generate network traffic in simulations, but is very difficult to compute for real programs except in synchronized parallel loops.

6.5. Parallel Execution Time of the Program

The execution time of a program on a multiprocessor system can be broken down into four categories: computation cost, data synchronization cost, control synchronization cost, and control overheads. The computation cost is the time the processor spends in doing the computation. The data synchronization cost is the length of time that a processor is forced to idle while waiting for needed data to arrive. Two sources of the data synchronization are local data accesses that cannot be overlapped with the computation and the synchronization introduced by the data dependence between a pair of statements that are executed by different processors. The cost of a data reference can be broken down into the *memory fetch cost*, which is the time taken for the request to reach the memory, and the *data transition cost*, which is the cost of transmitting the data through the network or bus. The control synchronization is the time it takes to synchronize the control between two or more processors (semaphores, barrier synchronization, conditional or unconditional branch statements, etc.). The control synchronization is usually introduced by explicit control structures, while the data synchronization is normally defined implicitly by the data dependence relations. The control overhead is the overhead for concurrent execution; this includes factors such as vector start-up time, process start-up time,

and dynamic task scheduling time, etc.

Based on the above discussion, a sample estimation of the execution time, T, of a program structure can be defined as:

$$T = T^{c} + T^{ds} + T^{cs} + T^{co} = T^{c} + T^{da} - T^{do} + T^{cs} + T^{co}. \tag{6.4}$$

where T^{c} is the computation cost, T^{ds} is the data synchronization cost (which is the difference of the data access cost (T^{da}) and the data overlapping time (T^{do})), T^{cs} is the control synchronization cost, and T^{co} is the control overhead. This is only one of the many different estimations that can be defined.

6.5.1. Example of Estimating Program Execution Time with the Model

The execution time of the program can be estimated to different degrees of accuracy with different sets of evaluation functions. The selection of the evaluation functions and the method for integrating the functions are done by a set of rules and depend on how much computing resources can be allocated to the estimation.

In this section, we use a simple example to illustrate the performance prediction process under our framework. We chose to use the nCUBE 2 to demonstrate how the estimation is established because the nCUBE 2 has some interesting characteristics that present a challenge to the performance prediction system. For example, integer addition and subtraction takes one machine cycle, but an operand decoding and fetching takes up to 3 cycles; array address calculation is expensive (the code generated by the nCUBE 2 compiler takes about 11 cycles for one dimensional arrays and about 31 cycles for 2 dimensional arrays); integer multiply (7 cycles) is slower than floating point multiply (6 cycles); integer division (38 cycles) is much slower than floating point division (7-18 cycles); communication between processors is very expensive (156 cycles start-up time per message for one hop), etc. These factors imply that data prefetch has dominant effects on the cost of the computation; simply classifying operations into floating point and integer operations may not be fine enough, and interprocessor communication needs to be overlapped with computation as much as possible. All these factors plus the fact that the compiler on the nCUBE 2 does not generate optimal code complicate the estimation of the performance on the nCUBE 2. Nevertheless, this presents a good opportunity for examining our performance prediction model.

The program that we chose is the matrix-vector multiply program as shown in figure 6.2. We assume that the program has been previously decomposed to be executed on P processors and that arrays a and y have been distributed by blocks across the processors, a local copy of x is available on all processors, and the result y is to be collected at processor 0. This example is chosen because it is simple but presents some communication imbalance for distributed-memory computers because processor 0 is a hot-spot. The purpose of the example is not to show how accurate the estimation can be since this varies with programs, but to show the process for constructing and refining estimations under the control of a rule base.

During the performance-setup process, the following performance factors are selected based on the architectural features of nCUBE 2: counts for floating point operations, integer operations, memory loads, and memory stores, array references, data synchronization, data transmitting cost, and control synchronization cost. All of the above factors except the data transmitting cost are in the list of default evaluation functions. The data transmitting cost over the network is needed because the nCUBE 2 is a distributed machine and the balance between the communication and computation is very important. The results of the evaluation functions will be combined at the task level. The performance of the program based on this setup is then:

$$\begin{aligned} T^{computation} = {} & (N^{fop} * C^{fop} + N^{iop} * C^{iop} + N^{ar} * C^{ar} \\ & + N^{load} * C^{load} * p^{ds} + N^{store} * C^{store} + N^{lo} * C^{lo}) * m * \ln^{i} \end{aligned} \tag{6.5}$$

where N^{fop}, N^{iop}, N^{ar}, N^{load}, N^{store}, and N^{lo} are the counts for floating point operations, integer operations, array references, memory loads, memory stores, and loop overheads, respectively, in the loop instance (k,l). Also C^{fop}, C^{iop}, C^{ar}, C^{load}, C^{store}, and C^{lo} are the unit costs for a floating point operation, integer operation, array reference memory load, memory stores, and loop overhead, respectively.

The constants that are used by the evaluation functions are provided by the machine knowledge manipulation mechanism and are listed in the following table:

A simple tree traversal reveals that there are 2 floating point operations, 3 array address calculations, 9 load operations and 1 store operation in the loop instance (i,j). Since the loop bounds of loops i and j are unknown, the evaluation functions are computed from the variables ln and m which are the loop bounds of

```
procedure matrix_vector_multiply(ln, m, a, x, y) returns (mbuf);
begin
forall pid in 1 .. P do          --- ln := ⌈n/P⌉.
      local a : array [1..ln, 1..m] of real;
            x : array [1..m] of real;
            y : array [1..ln] of real;
      for i := 1 .. ln loop            -- the simple part
            for j := 1 .. m loop
                  y[i] := y[i] + a[i,j] * x[j];
            end for
      end for
      if (pid == 0) then        -- the harder part
            received = 0;
            for i := 1 .. P-1 loop
                  wait_for_message(source, type);
                  recv(source, mbuf[source*ln], mbufln, flags);
            end for
      else              --   send vector y to processor 0;
            send(0, y, ln, flags);
      end if
end forall;
end procedure;
```

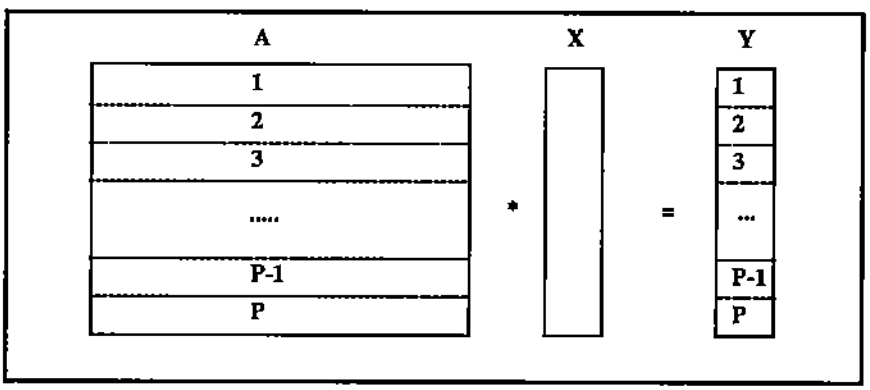

Figure 6.2. *Sample program and the array distribution for the arrays in the matrix vector multiply example.*

Table 6.2. *Constants for evaluation functions for nCUBE 2.*

Evaluation constants for nCUBE 2 (in microseconds)			
performance factor	*time*	*performance factor*	*time*
floating point op, C^{fop}	*0.33*	*integer op* C^{iop}	*0.35*
local memory load, C^{ld}	*0.11*	*local memory store* C^{ls}	*0.1*
message startup cost, α^0	*158.45*	*message startup (per hop),* α^1	*1.293*
data trans.(per word), β^0	*2.49*	*data trans.(per word per hop),* β^1	*0.0*
loop overhead, C^{lo}	*0.6*	*0 bytes read*	*80*

loop *i* and *j*, respectively. There are totally 2**m***ln* floating-point operations, 3**m***ln* array address calculations, 9**m***ln* load operations, and *m***ln* store operations for loop *i*. Also, the loop overhead for each processor is *m**ln times the unit cost for each loop instance. Since *m* and ln are both unknown to the procedure at compile time, the resulting estimation is a function of *m* and ln. The execution time of the loop *i* in each processor is then estimated by the performance construction model to be the following expression:

$$T^{computation} = (2 * C^{fop} + 0 * C^{iop} + 3 * C^{ar} + 9 * C^{load} * p^{ds} + 1 * C^{store} + 1*C^{lo}) * m * \ln \quad (6.6)$$

where C^{fop}, C^{iop}, C^{ar}, C^{load}, C^{store}, and C^{lo} are the unit costs for a floating point operation, integer operation, array reference memory load, memory stores, and loop overhead, respectively. And p^{ds} is the percentage of data that cannot be overlapped with computation.

For processor P, ln is $n - (P-1) * \left\lceil \frac{n}{P} \right\rceil$, and for all other loop instances, ln is $\left\lceil \frac{n}{P} \right\rceil$. A rule in the knowledge base suggests that for loops with only one statement in the innermost loop, data prefetching has very little effect unless the expression in the statement is very long. Since the statement in the inner-most loop, j, of figure 6.2 is relatively simple, the data synchronization overhead is set to be 100% of the memory load cost. Suppose N is 6400, M is 100, and P is 64, then ln = 100. By substituting the constants in table 6.2 into the formula, the estimated computation time for the loop k is 4600 microseconds. As a comparison, the measured computation time for loop i is 4823.6 microseconds.

The only cross-task data dependence in the program is caused by the communication statements corresponding to the collection of the array y, so the data-synchronization is computed with these statements. The control synchronization is hard to predict on the nCUBE 2 because there is a non-trivial cost in loading the node programs so most programs do not synchronize the program unless it is necessary. This means that tasks on the nodes start at different times depending on how the programs are loaded (broadcast or not), the size of the node program, and the position of the node in the hypercube. The control synchronization is needed only when values of y are needed later (which is out of the scope of the program that we discuss here). Lacking the control synchronization at the beginning of the program makes the estimation of the data synchronization cost difficult since tasks are not started at the same time.

To estimate the cost of sending messages, we need to estimate the cost for read/write and data transmission. Since message sending on nCUBE 2 is non-blocking, on tasks that send the data only the time to copy the data into system buffer needs to be counted. On the other hand, on the receiving side, the data synchronization cost includes the time for the sender to send out the data, the transmitting time, and the time for the reader to receive and store the data. This time is a function of the message length and the distance between the processors, and can be computed by the equation (6.7) below.

Since all the data is sent to processor 0, the processor 0 becomes a hot spot and the computation time of processor 0 dominates the computation time of the program. For processor i, the distance to processor 0 is the number of 1's in its binary representation of the processor identified, represented as $dist^0(i)$. Therefore, the cost of transmitting the vector y computed in processor i to processor 0 is:

$$T^{dt}(i) = \alpha^0 + \alpha^1 * dist^0(i) + \ln * (\beta^0 + \beta^1 * dist^0(i)) \quad (6.7)$$
$$= 158.45 + 1.293 * dist^0(i) + 2.492 * \left\lceil \frac{N}{P} \right\rceil$$

Although the receive statements in processor 0 are executed sequentially, the data synchronization cost is not the sum of costs of transmitting vector y from all other processors because the data is transmitted simultaneously. If all processors are synchronized, then the data synchronization cost would be:

$$T^{dt} = \max\left\{ \min\left\{ T^{dt}(i): i \in [1..P] \right\} + \sum_{i=1}^{P} T^r(i), \max\left\{ T^{dt}(i)+T^r(i): i \in [1..P] \right\} \right\}$$

where $T^r(i)$ is the cost of moving the message from the system buffer into the data structure of the user program.

For our sample program, $\min\{T^{dt}(i): i \in [1..P]\} = 159.743 + 2.492 * \left\lceil \frac{N}{P} \right\rceil$, $\max\{T^{dt}(i): i \in [1..P]\}$ $= 158.45 + dim * 1.293 + 2.492 * \left\lceil \frac{N}{P} \right\rceil$, and $T^r(i) = 80+0.11 * \left\lceil \frac{N}{P} \right\rceil$. We also add a term 60*P to conpensate the hot spot effect.

By substituting different values of n, m, and P, we obtain the estimations as shown in table 6.3. The measured execution times for these parameters are also shown for comparison.

Table 6.3. *The predicted and measured performance of the matrix-vector multiply program.*

			Predicted performance			Measured performance			% off
n	m	p	Computation	Communication	Total	Computation	Communication	Total8	
400	200	2	122800.00	981.74	123781.74	123632.00	729.00	124361.00	0.47
400	200	4	61400.00	1012.74	62412.74	61806.00	776.00	62582.00	0.27
400	200	8	30700.00	1448.24	32148.24	30909.00	1249.00	32158.00	0.03
400	200	16	15350.00	2505.99	17855.99	15475.00	2270.00	17745.00	0.63
400	200	32	7675.00	4714.87	12389.87	9391.00	3004.00	12395.00	0.04
400	200	64	3837.50	9179.31	13016.81	6977.00	5899.00	12876.00	1.09
1600	100	4	122800.00	1891.74	124691.74	124623.00	2008.00	126631.00	1.53
1600	100	8	61400.00	1953.74	63353.74	62179.00	2038.00	64217.00	1.34
1600	100	16	30700.00	2824.74	33524.74	31082.00	2990.00	34072.00	1.61
1600	100	32	15350.00	4940.24	20290.24	15564.00	5022.00	20586.00	1.44
1600	100	64	7675.00	9357.99	17032.99	7792.00	9266.00	17058.00	0.15
6400	20	16	24560.00	4099.74	28659.74	26010.00	6514.00	32524.00	11.9
6400	20	32	12280.00	5841.74	18121.74	13008.00	7632.00	20640.00	12.2
6400	20	64	6140.00	10072.74	16212.74	6516.00	36930.00	43446.00	62.7

The performance prediction established in the performance-construction module provides a foundation for estimating the effects of program transformations on the program. During the program optimization process, we can use this estimation and apply the performance-update module to adapt the estimation based on the program transformations. For example, in the above program, y[i] has an output dependence and a flow dependence on itself in loop j. This means that the result of y[i] is loaded and updated right after it is stored in the previous loop iteration. So if we allocate y[i] into the register during the execution of loop j, then only one load and one store of y[i] is needed. This is a reduction of $m-1$ loads and $m-1$ stores for each y[i]. Therefore, by allocating y[i]'s into registers, the load and store counts are each reduced by $(m-1)*ln$.

Note that the performance of the matrix-vector multiply program is far short of the advertised peak performance of the nCUBE 2. This is because there is only one statement in the loop body and the loop branches prevent the prefetching mechanism of the nCUBE 2 from pre-loading the operands. Consequently, the computation of the operands dominates the computation time. By applying the transformation *loop unrolling*, we may increase the size of the code between the conditional branch of the loop, thus allowing the data loading to be overlapped with the computation. For example, if we unroll the loop m 5 times, the estimated data fetching time becomes one-fifth of the original cost since p^{ds} is now 1/5. The loop overhead is also decreased by a factor of 5. This implies that by applying loop unrolling the cost for loading can be decreased significantly. This is confirmed by the result as shown in table 6.4:

Note that in tables 6.3 and 6.4, the estimation of the computation is fairly close to the actual cost of the computation. Although there are instances where the simple-minded estimation for the effects of the data prefetching is not accurate enough, this model can be utilized by the program optimization process to estimate the effects of program transformations on the performance. More detailed estimation of the data prefetching effects is needed only when this factor is important, for which case, the performance-refine module can be used to improve the accuracy of the estimation. On the other hand, the performance estimation for the communication is good for some cases but far from accurate in some other cases. This is because we did not consider the hot-spot effects of the message passing in the example. Since all nodes are sending messages to node 0, node 0 becomes the hot-spot, and when the message is long and the size of the cube is large, performance

Table 6.4. *The predicted and measured performance of the matrix-vector multiply program with innermost loop unrolled 5 times.*

			Predicted performance			Measured performance			%off
n	m	p	Computation	Communication	Total	Computation	Communication	Total	
400	200	2	71128.00	981.74	72109.74	69332.00	730.00	70062.00	2.92
400	200	4	35564.00	1012.74	36576.74	34691.00	889.00	35580.00	2.80
400	200	8	17782.00	1448.24	19230.24	17357.00	1247.00	18604.00	3.37
400	200	16	8891.00	2505.99	11396.99	8689.00	2261.00	10950.00	4.08
400	200	32	4445.50	4714.87	9160.37	5881.00	3004.00	8885.00	3.10
400	200	64	2222.75	9179.31	11402.06	5127.00	5899.00	11026.00	3.41
1600	100	4	71128.00	1891.74	73019.74	69898.00	1972.00	71870.00	1.60
1600	100	8	35564.00	1953.74	37517.74	34977.00	2056.00	37033.00	1.31
1600	100	16	17782.00	2824.74	20606.74	17499.00	2950.00	20449.00	0.77
1600	100	32	8891.00	4940.24	13831.24	8758.00	5065.00	13823.00	0.06
1600	100	64	4445.50	9357.99	13803.49	4406.00	9266.00	13672.00	0.96
6400	20	16	14225.60	4099.74	18325.34	14913.00	6478.00	21391.00	14.3
6400	20	32	7112.80	5841.74	12954.54	7468.00	7703.00	15171.00	14.6
6400	20	64	3556.40	10072.74	13629.14	3761.00	37045.00	40806.00	66.6

degradation occurs. Especially for the case where (n, m, P) = (6400, 20, 64), the cost of sending 64 messages of 100 words each to node 0 is more than five times that of sending 32 messages of 200 words each to node 0. Since the nCUBE 2 uses the *worm–hole routing* mechanism where a channel is reserved for a message and the data is pipelined to the destination, messages that need to use the busy links will be blocked until the previous message is finished with the channel. The longer the messages, the greater the chances that a message will be blocked at the sender side.

When a more accurate performance prediction is required, the performance-refine module can be invoked. For example, one way to refine the estimation is to use a finer classification of the operations. This will not help much for our sample program here since it does not contain any expensive operations such as divisions. For distributed architectures like nCUBE 2, one way to better estimate the data synchronization cost is to analyze the performance degradation due to the communication hot-spot. The performance degradation of the network caused by the hot-spot saturation can be estimated by the hot-spot percentage. However, the computation of the hot-spot percentage is very expensive and is generally not accurate due to the dynamic behavior of the program. A more practical approach is to model the performance degradation with a set of selected patterns of different degrees of congestion (as described by the data dependence graph). For data dependence that does not match any of the pre-selected patterns, an interpolation to the nearest pattern is done to find an estimation of the degradation. The more patterns we use, the better the estimation will be. However, matching complex patterns is an expensive operation, so this method can be used only when the user can afford the needed computing resources. For our example, the compile can easily recognize that processor 0 is a hot-spot and adjustment for hot spot effects can be applied.

6.6. Applying Performance Prediction to Intelligent Decision-Making

The performance prediction model can be applied to intelligent parallel compilers in the following areas:

1. To choose among a list of prospective transformations.
2. Provide a measure for selecting pre-optimized algorithms when several algorithms are available under the algorithm substitution approach.
3. Decide when to stop the optimization model.

4. Help to generate minimal run-time tests for situations where the compiler cannot make decisions by static analysis.

The performance prediction model we have proposed is designed to be integrated into the decision-making process for optimizing program parallelism. In particular, it is used in the *feature-directed program optimization model* to guide the decision-making process. When applied to systematic decision-tree traversing algorithms (such as the A^* algorithm as we discussed in chapter 4), automatic parallel program optimization can be achieved. Furthermore, the flexibility of our model makes it easy to refine the heuristic functions at different parts of the decision tree, which makes the framework more dynamic. It can also be combined with rule-based systems to improve the quality of the optimization. An evaluation function or the combination of several evaluation functions can be used to decide the effects of program transformation techniques. Heuristic driven rules can use the prediction to select appropriate program transformations or pre-optimized algorithms. In an interactive program optimization session, it provides users with performance information for them to make optimization decisions.

For a pre-optimized algorithm substitution approach, there are cases where difficult decisions need to be made. For example, there may be more than one algorithm that are equivalent to the program under consideration; or the same algorithm may be optimized for several different parallel architectures but not for the current target machine. For the former case, the performance prediction may help the system to decide which algorithm is most efficient for the problem. For the latter case, the performance prediction model can be integrated with the machine knowledge manipulation system to find the architecture that fits the target machine most.

In the program optimization process, the optimal solution for the program optimization is usually not known until the entire decision tree is traversed. This makes deciding the termination condition difficult. Since the performance estimation mechanism can be used to find the lower bound of the execution time, this number can be used to decide the termination condition of the optimization. First, a tolerance for the lower bound can be selected; the greater the optimization degree is, the smaller the tolerance will be. When the estimated performance falls into the tolerance range of the "optimal execution time," the optimization process can be terminated.

The compiler may not be able to make decisions based on the performance estimation which has uninstantiated variables in the expression. In these cases, multiple branches [BDHLW87] of the control may be generated, and when dynamic decisions are needed, these conditions of the performance with uninstantiated variables are used to find and generate the minimum run-time tests to decide the control flow.

Although any performance prediction model can be applied to the above tasks, our model is more flexible and efficient and can be easily integrated into the decision-making process of parallel compilers.

6.7. Related Work

There are many existing performance prediction tools for parallel architectures. For example, Faust [GGJMG89] and IPS [MiYa87] allow program behavior to be described at many levels of detail and abstraction including the program, process, procedure and instruction levels. PIE, developed at CMU [SeRu85], uses a metalanguage to provide support for an efficient manipulation of parallel modules and programming for observability. These systems are designed to be used as a semi-automatic performance evaluation tool for the user. The Parafrase system [AbKw85] provides a performance prediction module based on a similar program hierarchy but is designed to be used in the compiler and is inexpensive to compute. The "load/store" modeling method is used in [GJMW89, BWJALG90] to characterize the performance of shared memory architecture by a set of template sequences of vector load, store and "nop" instructions. There are many other environments that evaluate the performance of a particular type of architecture or characterize the potential parallelism of the program on the architecture. However, none of the existing systems can be used to predict program performance for a wide range of architectures accurately and inexpensively. Also, none of the systems provide a systematic framework that is flexible enough to be utilized in an intelligent parallel compiler. Our framework fills this gap by providing a flexible mechanism for the knowledge base system to adjust the prediction model dynamically to suit different optimization objectives and different architectures. [Leung90] derived several equations for static compile-time estimation of the overhead costs and execution time for several shared-memory machine models. However, our work is more general, flexible, and accurate.

6.8. Conclusion

To summarize, our framework for the performance prediction of parallel computers has the following advantages:

1. Different classes of parallel computers can be handled under the same framework. The performance prediction model can be adjusted to suit different architectures.
2. The estimation can be tuned to suit different objectives by adjusting the performance combining procedure and the weights associated with the evaluation functions.
3. Different amounts of resources can be committed at different stages of the compiling process by using evaluation functions of different complexities.
4. The selection of evaluation functions and weights offers a good opportunity for the compiler to learn to improve itself. This point will be investigated in [Wang91].
5. The prediction update process is inexpensive; only the effects of the related transformations are computed. It can be applied repeatedly during the program optimization process.

Intelligent parallel compilers need an accurate, inexpensive, and flexible performance prediction model to make critical decisions. Our framework is simple and yet flexible enough to be integrated into intelligent parallel compilers with multiple target machines. It can be used by systematic state-space search algorithms such as A^* and other best-first search algorithms to find optimal program transformation sequences. It can also be used by heuristic-driven rule-based systems to choose the program transformations at compile time.

CHAPTER 7
ABSTRACTING PARALLELISM FOR MIMD PARALLEL COMPUTERS

The key issue in optimizing a program for a target architecture is to match the program parallelism with the available machine parallelism. Chapter 4 discussed different approaches for systematic analysis and inference to match the program and target architecture. In this chapter we discuss methodologies based on performance estimation, evaluation functions and heuristics to support this matching process. The heuristic functions discussed in the next section quantify the match between the program parallelism and the machine parallelism and form the basis for performance estimation and program optimization.

7.1. Improving Parallelism with Feature-Directed Program Optimization

In this section we discuss the methodologies and heuristics for program optimization based on the feature-directed program restructuring model.

The program optimization process first decomposes the program into units of execution call tasks. To parallelize a program or increase the concurrency of a program, the program is restructured for concurrent execution by converting loops into parallel loops and executing tasks concurrently. The tasks and inter-task dependence relations form a graph that is called the task graph. To preserve the correctness of the program, synchronization is needed for concurrent execution. There are two kinds of synchronization: *explicit synchronization*, defined by synchronization statements in the control flow (such as barrier synchronization, conditional statements), and *implicit synchronization*, defined by the data dependence relation between a pair of tasks allocated to different processors.

For shared-memory machines, explicit synchronization instructions such as locks, shared semaphores, protection bits or words, etc. are generated for inter-task data and control dependencies. For do-across loops, do-across delays [Cytron84] are also inserted to minimize the stress of constant polling at the shared variable on the network and memory modules.

For distributed-memory machines that use a message passing paradigm, the inter-task dependence relations are replaced by explicit message read/write instructions that transmit the data between processors. The overhead for sending and receiving messages is usually very high (when compared to computation, with the exception of Pringle, in which sending a word costs about the same as a floating point operation). Thus, minimizing the number of messages is of major concern.

7.1.1. Focus of Program Optimization

Since a program is usually fairly large, it is divided roughly into modules that we call program focuses. A program focus is a program unit (usually loops or a large block of code) that is the central focus of the parallelism-optimization. Program focuses usually define a sequential thread of executions. However, focuses can be merged or the target machine can be decomposed and parts of the machine assigned to different focuses for concurrent execution.

One major reason for decomposing a program into program focuses is for better utilization of the limited resources. For most programs, the major part of the execution time is spent in a small portion of the program; therefore, concentrating limited computing resources on the most important program focus is critical to the success of the compiler. A program focus which is small in size also makes the optimization process easier. The decomposition of the program can be guided either by profiling or by heuristic evaluation functions. The profiling of a program can usually expose the most computation or communication intensive portion of the program for programs whose performance is not sensitive to input data. The advantage of using heuristic

evaluation functions is that the results of the evaluation functions can be functions of variables whose values are only available at run time. This allows the generation of multiple thread controls that can be decided at run time based on the values of those variables. Also, heuristic evaluation functions such as the operation counts are simple to compute and often do a fairly good job without the program actually being executed.

Our heuristic for selecting focuses in a program is as follows:

1. Use profiling to find the few subroutines or functions that take the longest execution time.
2. The top level loops in these subroutines form focuses of their own. Statements between these loops also form focuses.
3. After the focuses are decided, they are ordered on the basis of heuristic evaluation functions such as operation counts or the estimated execution time.
4. The program optimization process is applied to each of the focuses starting from the program focuses with highest priorities until the computer resources are exhausted. The allocation of the computer resources can be determined by heuristics based on the optimization degree specified by the user or dynamically determined by the estimated execution time of the focus.

Decomposing a program into a set of focuses has the following advantages:

1. It is easier to optimize since the size of the program focuses is much smaller than the original program.
2. The resources are spent on the most important part of the program.

The disadvantage of this approach is that local optimization of the focuses may produce conflicts among focuses. To solve this problem, a global optimization phase to resolve conflicts between the focuses can be applied after all the focuses are optimized. Our approach of ordering the focuses also serves to prevent some of the problems since conflicts can be avoided by respecting the decisions made by the optimization on the most "important focus" before the current focus is selected.

7.2. Heuristic Guided Program Optimization

A hierarchical framework for feature-based program optimization was discussed in section 3.4.2. This framework classifies the program optimization process into the following five groups: general program parallelism optimization, task decomposition, processor allocation and assignment, memory utilization, and synchronization minimization. Based on the hierarchical problem decomposition, each of these five processes utilizes a different set of program transformation heuristics to explore different aspects of the concurrency. These five processes cooperate to achieve global optimization. The architecture of the hierarchical-blackboard allows non-determinism to be exploited at different levels of the hierarchy in the form of knowledge sources. We will discuss some methodologies for program parallelization and optimization based on the hierarchical structure of the program optimization process. Our methodology for controlling the decision flow is to use heuristic-driven rules at the upper levels of the hierarchy and use the hierarchical blackboard control to exploit the concurrency and non-determinism at the lower levels.

7.2.1. General Optimization of the Program Parallelism

The first step in program optimization is to apply machine-independent program restructuring transformations to expose all the parallelism inherent in the program and thus improve its potential for further machine-specific optimization. General program optimizations include dependence cycle breaking, locality improvement, and eliminating redundant instructions. Breaking program dependence cycles or eliminating dependence relations is always beneficial because a program dependence creates synchronization delays and a dependence cycle forces the statements in the cycle to be serialized. For most parallel architectures, better locality means faster data accesses and fewer processing delays. For these reasons, a fixed sequence of applicable machine-independent program transformations can be applied to the program focus. The following is a list of program optimization techniques that are applied to improve the parallelism of the program.

- Scalar expansion: breaks output- and anti-dependence associated with the expanded variable.
- Variable renaming: breaks output- and anti-dependence associated with the expanded vector.
- Statement splitting: breaks the dependence cycle by repositioning the anti-dependence arcs.
- Forward substitution: removes flow dependencies associated with the expression.

- Statement reordering: moves statements out of dependence cycles, improves locality, etc.
- Array gathering: compresses the array to improve locality.
- Array reshaping: reshape the array to improve locality (discussed in the next section).
- Loop interchanging: improves locality
- Code motion: moves loop invariant code outside of loops.
- Dead-code elimination: eliminates redundant code
- Loop merging: saves loop overhead.

7.2.2. Task Decomposition and Processor Assignment

Program fragments that are executed as a unit sequentially are called *tasks*. A task is the basic unit for processor allocation and scheduling. To decompose the program focus into a series of sequential tasks, the following simple heuristic is used:

Heuristic 6.1. (Task definition)

1. Instances of the top level loops form tasks by themselves.
2. Conditional or unconditional branch statements (if and exit statements) form tasks.
3. All other statements form a task of their own. We then apply topological sorting to these statements based on the program dependence graph and the strongly connected components in the program dependence graph form tasks.

The above heuristic decomposes the program into a graph of tasks connected by inter-task dependencies. Statements in strongly connected components are grouped into a task because dependence relations require control and data synchronization. This implies that the group of statements has little concurrency and gives rise to the following heuristic:

Heuristic 6.2. The communication cost resulting from executing a strongly connected component on multiple processors usually overshadows the benefit from the concurrent execution.

Tasks that are not top-level loops can be assigned to processors by a high-level task-spreading algorithm. For example, there is a task composition algorithm TACOM in [Poly86] that merges tasks on the basis of a shared-memory model. The algorithm merges two tasks when their concurrent execution causes large communication costs. Since loops have regular patterns, contain abundant parallelism, and are easier to parallelize, they are usually handled differently than other tasks. A loop is parallelized when its loop instances are selected as parallel tasks. The selection of parallel loops is usually based on the estimated parallel execution time.

The task assignment and memory utilization form a classical chicken-and-egg problem; the optimal task assignment must take the memory access time into consideration, and the estimation of memory access time is only possible when the tasks are allocated. In our approach, we take a two-phase approach for the task creation and allocation problem. The tasks are first created by using some criteria such as the ones listed in heuristic 6.1, and are adjusted by an algorithm such as TACOM or a parallel-loop selection algorithm. Then the memory access optimization and synchronization minimization is performed based on the tentative tasks and processor assignment. During the program optimization process, the tasks may also be merged to form larger tasks or be further decomposed into smaller tasks, depending on the requirements of the current goals. The task composition and processor assignment are finalized after this process. This two phase approach is necessary because optimizing memory access and minimizing synchronization need to be based on the task assignments, but, on the other hand, optimal task assignment needs to have good estimation of the execution time of the restructured program to balance the computation load.

7.2.3. Memory Utilization for Shared-Memory Architectures

In this section we investigate some heuristics and techniques for optimizing the memory utilization on shared-memory architectures. In [Husm86] Husmann uses the following heuristics for array allocation on shared-memory machines:

Heuristic 6.3. Assuming the tasks have been created and allocated, the following heuristics are used to allocate the arrays.

1. An array can be allocated to a local memory only when it is neither declared as a global variable nor referenced by multiple CPUs.

2. An array can be allocated to cluster memory only when it is neither declared as a global variable nor referenced by CPUs in multiple clusters.
3. All other arrays are allocated to the global memory.

This heuristic is simple to implement and it can be used by the task creation and assignment algorithms to estimate the data reference and program execution time. However, this approach is very conservative and ignores many possible ways of speeding up the data references. For example, a more aggressive data allocation strategy involves breaking down the array and allocating portions of the array to different processors. Another approach is to copy a subset of the array into local memory and use the local copy instead of the global. Furthermore, [Husm86] assumes that the costs of all memory operations, data accesses, and computation are constant, which forms the basis of the estimated execution time used to select the parallel loop. This assumption (especially that the costs of network accesses and memory references are constant) is unrealistic and affects the accuracy of the result. Data block-access is used in Husmann's algorithm but only under the condition that no multiple PE write the same elements of a block and a PE not write to an element in multiple blocks. Also, multiple level parallel loops and interactions of array references in different blocks are not considered. Husmann's algorithm can be used to generate initial data allocation and other heuristics can be used to improve the performance. We will list some extensions here.

- Copy repeatedly used shared arrays (such as the variable x in the matrix-vector multiply problem) into local memory to decrease network traffic and increase locality for target machines that have no cache.
- Use the transformation *array reshaping* discussed in the next section to move a subset of an array to local memory to increase locality.
- When any part of an array is updated by multiple processors, Husmann's algorithm will not attempt to block-transfer the array to local memory. We can use array reshape to separate the array into two parts with one half containing elements that are block-transferable and the other half containing elements that are not block-transferable (elements that are updated by multiple processors or by a single processor but in different blocks).

7.3. Array Reshaping -- a Mechanism for Optimizing Array Usage

In this section, we introduce a program transformation technique called *array reshaping*. Array reshaping is a program transformation technique to modify the storage pattern of an array (called the *shape* of the array). The shape of an array is the way the elements of the array are physically stored and is defined by the declaration of the array. This transformation is not to be confused with some existing transformations such as index shifting, loop skewing, and linearization, that change the *view* of an array. A *view* of an array is an index set of the elements of the array and is a function defined by subscripts of an array reference. An array can have many views but only one shape. When an array reference appears inside loops, the subscripts of the array reference are functions of indices of the loops. Program transformations can change the view of an array by changing array subscripts or loop indices. For example, an n by n tri-diagonal matrix with only three non-zero diagonal elements has a square shape of size n by n . By altering the subscripts of the array reference in the loop nest, the view of the array can be changed and only the elements that are on the tri-diagonal need to be referenced. However, the change of the views does not affect the physical pattern of the array storage. If array reshaping (as shown in the example in the next section) is applied on the array, the shape of the array may be changed into a filled array with only non-zero elements of the original array, i.e. the array has a rectangular shape of size n by 3.

Definition 7.1 The *shape* of an array a can be characterized by bounds of the array and can be defined as:

$$shape(a) = [l_1 .. u_1] \times [l_2 .. u_2] \times ... \times [l_L .. u_L].$$

where l_k and u_k are the lower and upper bounds of the k-th subscripts of the array.

The *size* of the array is then defined to be:

$$Size(a) = \prod_{i=1}^{L} (u_i - l_i).$$

By nature, array reshaping is a global program transformation that can be applied only after global analysis of the program to determine which shape and storage pattern of the array can yield the best program performance or minimal program storage. Whether any elements thrown out by the reshaped array are used

anywhere in the original program also needs to be checked. These can be analyzed with the define-use chain of the program dependence graph and the program performance prediction model that we proposed in chapter 6. Another important usage of array reshaping is to identify the portions of arrays that are used by certain parts of the program and to make a local copy of these portions. This means that instead of the original array being replaced, the reshaped array is placed in the local memory of a processor and the code is generated to move the data from the original array to the new array.

7.3.1. Array Reshaping Functions

Array reshaping represents a group of one-to-one functions that operate on the shapes of arrays. A reshaping function defines how the storage of an array is to be changed as well as the relation between the elements of the original and the generated array. There are two kinds of array reshaping that we are particularly interested in: truncation/extension and linear transformation. Truncation changes the lower and upper bounds of the subscripts by removing or adding spaces to the arrays. The extension extends the boundaries of an array beyond its original bounds by enlarging the bounds of its subscripts. Linear transformation applies linear functions to the subscripts of the array and changes the shape of the array. The difference between these two types of reshaping functions is that for linear transformation, the same linear function is applied to both subscripts of the elements and bounds of the array; whereas for truncation or extension, only the bounds of the array are changed (an identity function is applied to the subscripts of the elements). A truncation or extension is often accompanied by a linear transformation function.

A truncating or extending array-reshaping function (represented as $[l..u]_i$) that changes the bounds of the i-th subscript of an array a into [l..u], changes the shape of the array a from

$$[l_1 .. u_1] \text{ x } ... \text{ x } [l_i .. u_i] \text{ x } ... \text{ x } [l_L .. u_L]$$

into

$$[l_1 .. u_1] \text{ x } ... \text{ x } [l .. u] \text{ x } ... \text{ x } [l_L .. u_L].$$

A linear reshaping function $(f_1(I), f_2(I), \cdots, f_L(I))$ maps the element $a(i_1, i_2, \cdots, i_L)$ in the array a into $a^r(f_1(i_1,i_2,\cdots,i_L), f_2(i_1,i_2,\cdots,i_L), ..., f_L(i_1,i_2,\cdots,i_L))$, where a^r denotes the new array. And the shape of the array a is mapped from

$$[l_1 .. u_1] \text{ x } [l_2 .. u_2] \text{ x } ... \text{ x } [l_L .. u_L]$$

into

$$[f_1(\overline{l_1}) .. f_1(\overline{u_1})] \text{ x } [f_2(\overline{l_2}) .. f_2(\overline{u_2})] \text{ x } ... \text{ x } [f_L(\overline{l_L}) .. f_L(\overline{u_L})]$$

where $\overline{x_i}$ denotes a vector whose entries are all zero except the i-th entry, which has the value x. For convenience, the linear array reshaping function is denoted as:

$$I \longrightarrow (f_1(I), f_2(I), \cdots, f_L(I)).$$

7.3.2. Variations of Array Reshaping

Although array reshaping can be used to change the shape of array into many different shapes, the primary purpose is to reduce the size of the array. Only elements whose images of the reshaping function fall into the new shape of the array have slots in the new array. Some basic reshaping functions are listed as follows:

- *Projection: (I, j) --> (I) or (i, I) --> (I).*

  This function can be used when the interval corresponding to the dropped subscript in the shape of the array is trivial, that is, it has only one constant in the interval. If one subscript of an array a in a for loop is loop invariant, then the array can be projected into a lower dimensional array inside the loop. For two-dimensional arrays, the projection (i,j) --> (i) maps the original array into the i-th row and the projection (i,j) --> (j) maps the original array into the j-th column. The projection can be viewed as a special case of truncation; when the range of a subscript of the array is truncated into only one integer, the corresponding dimension of the array can be dropped in the task.

- *Transposing: (i,j) --> (j,i).*

  Two subscripts of the array are exchanged. In this case, an array of shape [1..N]x[1..M] will be changed into an array of shape [1..M]x[1..N].

- *Compaction:*

  A linear function that can compact the bounds of the array subscript is called compacting function. For example, (i) -> (i/2) reduces the size of the array in half.

- *Expansion:*

  A linear function that expands an array into a larger array is called an expansion function. For example, (i) --> (2*i) doubles the size of the array. The expansion is usually applied on arrays that were compacted to map them back to the original arrays.

  We denote compaction and expansion as:

$$a' \longleftarrow ((i,j)\longrightarrow(f_1(i,j),f_2(i,j)))(a)$$

$$\text{and } ((i,j)\longrightarrow(f_1^{-1}(i,j),f_2^{-1}(i,j)))\,(a') \longleftarrow a$$

  respectively, where f_i^{-1} denotes the inverse function of f_i.

  To reduce the data transmitting cost in distributed systems, for a compacting reshape the compacting is usually done before the data is sent; and for expanding reshape, the expansion is usually done at the receiving processor.

- *Integer linear functions:*

  An integer linear function is a linear function whose coefficients are all integers. A general form for the linear functions is $\sum_{k=1}^{n} a_k * i_k$, where a_ks are integer coefficients and i_ks are the indices of the array subscripts.

7.3.3. Opportunities for Applying Array Reshaping

"How can array reshaping be used in program restructuring to improve the parallelism of a user program on a target architecture?" Array reshaping can be used to copy the data into the local memory of a processor or change the storage pattern of an array to reduce space or the access time. The latter includes simplifying array subscript-calculation, improving cache-hit ratio, changing array strides without interchanging the loops and reducing unnecessary traffic on the network. We list a few cases here and study these cases through examples.

Case 1. Minimizing the array storage.

For example, consider a band matrix a of size n by n with width w declared as:

a: array [1 .. n, 1 .. n] of real;

The matrix uses n^2 spaces. By reshaping the band matrix into a rectangular array of size n by $2*w+1$, the storage requirement is decreased to $n*(2*w+1)$. This situation can be recognized by observing that the second subscript j of the array reference $a(i,j)$ is always bounded by $i-w$ and $i+w$. By transferring the array based on the function (i,j) --> $(i,\ j-i)$, the array a is mapped into a new array a' with the elements being repositioned. And the declaration of the array becomes:

a: array [1 .. n, -w .. w] of real;

Case 2. Changing the physical reference order of the array to improve performance.

For example, transposing a vectorizable array that has a long array stride but happens to be in a pair of non-interchangeable loops may yield a stride-1 vector. Even if the loop is interchangeable, we still have a case where in a doubly-nested loop we have two references to two arrays for which one has stride-n reference and another has stride-1 reference. By interchanging the loops, we would change one reference into stride-1 and another into stride-n.

The reshaping function can also be compounded with other reshaping functions to change both the view and shape of the array.

For example, for the illustration in case 1, if the array is better transposed (for it to be vectorized or for other purposes) for other parts of the program, the transposing function (i,j) --> (j,i) can be applied. By applying (i,j) --> (j,i) to the above example we obtain a combined reshaping function (i,j) --> $(j-i,i)$, and the array declaration becomes

a: array [-w .. w, 1 .. n-1] of real;

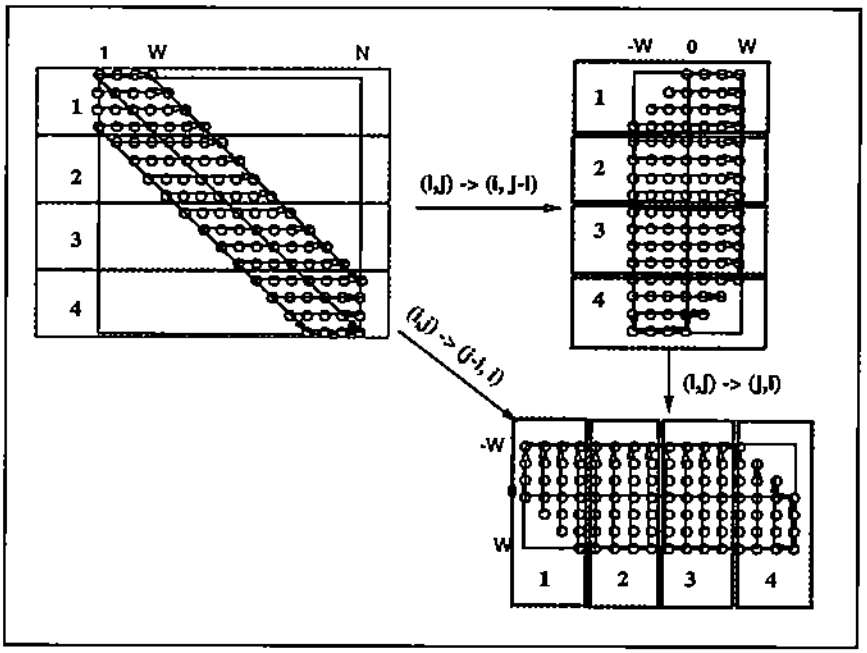

Figure 7.1. *Example of reshaping a band matrix.*

Case 3. Minimizing messages in distributed-memory systems.

The data communication between processors of distributed-memory systems can be significantly reduced when the sub-array that is used is compacted and copied over. If the sub-array is modified, the part of the array that is modified can be stored back to the original processor by transmitting the results back and expanded into the original array.

For example, suppose array *a* is stored in process *i* but used by processor *j* in the following loop:

```
for i := 0 .. P-1 do (* parallelized loop *)
    for k := 0 .. M-1 do
        b[i, k] := a[i-1, 2*k];
        ....
    end for
end for
```

In a distributed-memory system, the reference to array *a* will be replaced by a copy of the array *a* as shown below:

```
forall i := 0 .. P-1 do
    local aloc: array [ 0 .. M-1 ] of real;
    if (i != 0) aloc[0..M-1] <- a[i-1, 0..M-1];
    for k := 0 .. M-1 do
        b[i, k] := aloc[2*k];
    end for
    ....
end forall
```

Then array a has to be sent from processor i to processor j. By reshaping a', the copy of the array a in the processor j, with the reshaping function $i \longrightarrow (i/2)$ on the local array *aloc*, the program becomes:

```
forall i := 0 .. P-1 do
    local aloc: array [ 0 .. M/2 ] of real;
    if (i != 0) aloc[0..M/2] <- (a[i+1, 2*j], j=0..M/2);
    for k := 0 .. M-1 do
        b[i, k] := aloc[k];
    end for
end forall
```

As can be seen in the example, the conversion is done in the sender and the size of the array that is sent through the network is halved and the array subscript computation is also simplified.†

Note that this transformation is based on the specific shape and usage of the array; numerical analysts have been doing this by hand for decades. However, this hand optimization usually obscures the clarity of the algorithm and sometimes it also creates difficulties for the compiler in optimizing the program. Having the compiler perform these kinds of optimizations automatically not only eases the burden on the programmers but also makes the optimization easier on the compilers. This point is made even more clear in the following example:

Case 4. Avoiding obscured algorithms due to optimization.

Excessive program optimization often leads to obscured algorithms. For example, consider factorizing a band matrix using Gaussian elimination. The following program is a direct coding of the Gaussian elimination with the additional knowledge that array a is a band matrix.

```
a: array [1..n, 1..n] of real;
for i in 1 .. n-w do
    for j in i+1 .. i+w do
        a[j,i] := - a[j,i] / a[i,i];
        for k := i+1 to i+w do
            a[j,k] := a[j,k] + a[j,i] * a[i,k];
        end for
    end for
end for
```

For large n, most "space" in array a is unused and wasted. In order to save space, a "competent" programmer would implement the above algorithm as follows:

```
a: array [1..n, -w..w] of real;
for i in 1 .. n-w do
    for j in 1 .. w do
        a[j+i,-j] := - a[j+i,-j] / a[i,0];
        for k := 1 to w do
            a[j+i,k-j] := a[j+i,k-j] + a[j+i,-j] * a[i,k];
        end for
    end for
end for
```

Unfortunately, this program is so obscure that most people have to spend quite a bit of time to comprehend the meaning of the subscripts and the program. By examining the above example more carefully, we will find that the second program can be obtained by shifting the loop indices of loop j and k by i and then applying the reshaping function $(m,o) \dashrightarrow (m,o-m)$ to the first program. This optimization can be obtained by a simple heuristic encoded as rules in the rule base. By letting the compiler optimize the usage of storage, the program can be specified as close to the original algorithm as possible.

---

† The same effect may be obtained by using other transformations but array reshaping is a more powerful and more general transformation.

Case 5. Array Copying for the Functional Semantic of Forall Loops.

For languages whose *parallel* loops have functional semantics, copy-arrays may need to be created in each loop instance that uses or updates the array to preserve the copy-in and copy-out semantic [Wang86]. Array reshaping can be used to reshape the local copy-array of the original array for each loop instance. Since memory accesses inside for loops are usually very regular, this means that array reshaping can reduce the size of the copy-array to the minimum. This will guarantee that only the necessary data is copied into the parallel loops. On distributed systems, only the remote array elements that are used by the local processors need to be copied. This minimizes the cost of copying arrays in the implementation of functional FORALL loops and makes it practical.

For example, consider the Jacobi iteration shown in the next program fragment:

```
var
    A, New_A: array [0..N, 0..N] of real;
    pid: integer;

forall i in 1 .. N-1, j in 1 .. N-1 do
    New_A[i,j] := 0.25 * (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]);
end forall;
```

Note that this forall loop is usually surrounded by an outer loop that iterates until the solution converges. At the end of the iteration there is code to copy contents of the array New_A into A (or an optimization-minded user might interchange the array New_A with array A to avoid the copying). For shared memory architecture, the array A and New_A can be in global memory, but it is beneficial to create a local copy of the array. The local copies of the array Old_A are block-transferred to the processors before the first iteration and the results are copied back after the last iteration. This model is similar to the distributed-memory model. To execute this program on a distributed-memory machine, one simple decomposition is to partition the arrays into P^2 blocks of size MxM, where $M = (N + 2) / P$. This yields the following program:

```
forall k in 0 .. P-1, l in 0 .. P-1 do
    const
        low1 = k*M; low2 = l*M;
        high1 = min(low1+M-1, N+1); high2 = min(low2+M-1, N+1);
    var
        Old_A, New_A: array [low1-1..high1+1, low2-1..high2+1] of real;
    DATAMOVEMENT();
    for i in 1 .. M, j in 1..M do
        New_A[i,j] := 0.25 * (Old_A[i-1,j] + Old_A[i+1,j] +
            Old_A[i,j-1] + Old_A[i,j+1]);
    end for
end forall
```

In the above program, the array Old_A is the reshaped array of the original array A that the iteration uses for each process. There is an overlapping area of the array with processors that compute the adjacent blocks. If we only look at the forall loop, the array New_A should be declared as follows:

```
New_A: array [low1..high1, low2..high2] of real;
```

However, since the array Old_A is copied into (or switched with) array New_A, after the forall loop, the boundaries of the array New_A are extended (array reshaping replaced the above declaration by the extended array). Based on the dependence analysis of the original program, the statement DATAMOVEMENT() represents the code to move data into the local processor at the first iteration of the outer loop of the forall loop and moving data from adjacent processors and copying inside the processor in the subsequent iterations. This simple example actually represents a very sophisticated process of program restructuring.

7.3.4. Heuristics for Applying Array Reshaping

How does a compiler recognize opportunities for applying array reshaping? When is array reshaping beneficial? Since array reshaping changes the declaration of the arrays, it is only applicable when all the

expressions involved are resolvable at compile time. In the last section, we listed some opportunities for applying the array reshaping. Here we list some simple heuristics that a parallel compiler can use to decide when to apply array reshaping.

1. If a subscript of an array is a constant for all references of that array inside a task, then the projection can be applied to the array to map the array to a new array with the subscript dropped. Inside loops, the precondition means that the said subscript of all references to the array is loop invariant.
2. If the range of one subscript in all references of the array is a subset of the bounds of that subscript, then the shape of the array can be shrunk by truncation to truncate the bounds of that subscript into the actual range.
3. If a subscript of the array is a multiple of a loop index by an integer, such as $a*i$, then the array can be compacted by the array-reshaping function i/a in that subscript. This heuristic was used in the last example.
4. If the expression of a subscript appears in another subscript of the array, this expression can be eliminated by array reshaping. For example, for the program in the example in case 4 in the last section, the index of loop i appears in the loop bounds of loops j and k. Consequently, array references in the loop that use indices j and k are determined by the value of i implicitly. Just to see the relations more explicitly, we apply the index shifting transformation to the original band matrix factorization program shown in the above example and obtain:

```
for i in 1 .. n-w do
    for j in 1 .. w do
        a[j+i,i] := - a[j+i,i] / a[i,i];
        for k := 1 to w do
            a[j+i,k+i] := a[j+i,k+i] + a[j+i,i] * a[i,k+i];
        end for
    end for
end for
```

From the above transformed program it is clear that the index i appears in every subscript of every reference. For each loop instance of loop i, i is a constant for the loop instance. Therefore, the shape of the array a in the loop instance is:

$$[i+1..i+w] \times [i] \cup [i] \times [i] \cup [i+1..i+w] \times [i+1..i+w] \cup [i] \times [i+1..i+w]$$
$$= [i+1..i+w] \times [i+1..i+w].$$

The shape of all references for a in the loop is $[i+1..i+w] \times [i+1..i+w]$, where $1 \le i \le n$. This implies that the array references of a in the loop all fall into a band of width w. Our heuristic says that by applying the function $(i,j) \rightarrow (i,j-i)$ to the subscripts of the array reference, the shape becomes $[i+1..i+w] \times [-w..w]$. And for loop i, the shape of the array becomes $[1..n] \times [w..-w]$, achieving a reduction of $(n-2*w-1)*n$ in space. The second subscript in the new array represents the distance of the element to the diagonal in the original array.

7.3.5. Some Remarks for Array Reshaping

The methodologies that we described in this paper can serve as a starting point for studying array reshaping for program optimization. Array reshaping is a powerful but complicated program transformation technique. It is particularly useful for minimizing data communication cost for architectures that have non-trivial data transmission costs and programs that only utilize parts of the arrays. It can also be used to minimize data storage for machines that have limited memory available (such as the hypercube computers). Since array reshaping has significant effects on data references and communication costs, its usage should be carefully planned to avoid disastrous counter effects. The effects of array reshaping on the performance of the program depend on the architectural features of the target machine and can be estimated by the performance prediction model. This warrants a more thorough study about the potential benefit of array reshaping in optimizing programs for parallel machines, especially when combined with other program transformation techniques.

7.4. Message Consolidation

7.4.1. Introduction

Automatic program generation for distributed memory parallel computers is a very difficult problem and has been largely ignored until recently. Nevertheless, the difficulties in programming distributed memory parallel computers make this problem ever more important as the distributed memory architectures such as nCUBE 2 [NCUBE90], iPSC/2, iPSC/860 [Intel90], and Intel Touchstone, etc. become more powerful and popular. The major issue in programming distributed memory parallel computers lies in the distribution and communication of the data. One approach being followed by several groups is to support a global shared name space at the program level and to automatically generate the communications required for non-local references [CaKe88], [Koel90], [MeVR89a], [MeVR89b], [RoScWe89], [RoPi90], [RüAn90], [ZiBaGe88]. Although such a distributed shared memory approach provides users with a global memory space and allows them to program distributed memory systems in a style close to shared memory computers, the fundamental problem of reducing the communication and synchronization overhead remains to be solved by the underlying compilers. The techniques we describe in this paper can be used to minimize the communication and synchronization overheads in compilers that support program parallelization or distributed shared memory models.

For distributed memory architectures that use the message-passing paradigm, non-local data references need to be converted into explicit send/receive instructions. The simplest approach is to generate a pair of send/receive statements for each data dependence and utilize a set of control libraries that use message passing for control dependencies. The problem is that message passing is an expensive operation and the communication overhead along with the serialization effects of the send and receive operations might destroy any benefits of parallel execution.

One possible way of reducing the communication cost is to consolidate the messages into longer messages. This technique has been practiced by many parallel programmers in programming distributed parallel computers for a long time, but its use in parallel compilers for the automatic program optimization of distributed parallel computers has come into use only recently. This optimization is currently being studied by Several distributed memory compiler research groups are currently studying this optimization in the context of loops [CaKe88], [RoPi90], [Gemdt90]. The basic approach followed by these groups is to first spread the iterations of a loop across the processors. Any non-local reference within the iterations is then preceded by a send/receive pair so as to communicate the appropriate data element. Where possible, such communication statements are extracted out of the loop and then "vectorized" into a single communication statement. Thus, instead of each iteration generating its own message, a single message is utilized between any two pairs of processors to exchange the required non-local data. This optimization of messages in loops has been incorporated in some of the above compiler efforts, however, the theoretical foundations and technical details of this approach for general code have not yet been fully investigated.

Consolidating messages has two apparent effects: decreasing message passing overheads and increasing data synchronization delays. Whether two messages can be profitably consolidated depends on the tradeoff between the above two factors and the data dependence. Careless message-merging may slow down the program, generate incorrect results or cause communication deadlocks. In this paper we examine the tradeoff of consolidating messages and present an algorithm for deciding the optimal clustering of the messages.

7.4.2. Foundation of Message Consolidation

In the following discussion, we presume that a compiler has already generated the tasks to be executed asynchronously on a distributed memory machine. In such a situation, each cross-task data dependence gives rise to a data synchronization point so as to ensure the correctness of the concurrent execution. Enforcing these data synchronization points has a high overhead on distributed memory systems because data communication is a very expensive operation on these machines. For example, the cost of sending a 4-byte number to a neighbor on an nCUBE 2 processor costs about 160 microseconds as compared to a floating point multiply which costs only 0.35 microsecond. In order to minimize the overhead of data and control synchronization, the compiler can attempt to merge multiple messages into single longer messages and overlap communication with computation. In other words, a single data synchronization point for multiple data dependence between two tasks is preferred. Unfortunately, merging data synchronization points means delaying the startup time of the data transfer and this decreases the overlapping of the data transmission of the message with the computation at the receiving processor and thus increases the data synchronization cost. It is therefore necessary to

derive an algorithm that decides when and how data synchronization points can be merged beneficially.

Before we discuss the algorithm for performing such an optimization, it is necessary to examine some theoretical foundations for the approach. In the following discussion, we assume that the architecture supports read, write and test operations. The write statement sends the message, the read statement receives the message, and the test operation checks if the designated message has arrived. We further assume that the write is non-blocking and the read is blocking, that is, the sending processor can proceed with other computation after the message is transferred to the underlying network transport hardware, but the receiving processor will have to wait in the receive statement until the message has arrived.

In the following discussion, we assume that T_1 and T_2 are two tasks and δ is a flow or output dependence from statement S_1 in T_1 to S_2 in T_2. Let $t(S_1)$ be the execution time of the statements between the first statement in T_1 and S_1 *including that of* S_1, and $t'(S_2)$ be the execution time of the statements between the first statement in T_2 and S_2, *excluding that of* S_2. Let M be the size of the data that causes the data dependence, and $trans(M)$ be the cost of transmitting data of size M.

Lemma 7.1. The data synchronization delay in the receiving processor caused by the data dependence δ is given by:

$$delay = \max\left\{0,\ t(S_1) + trans(M) - t'(S_2)\right\} \tag{7.1}$$

Proof:

The data sent by task T_1 will arrive at task T_2 at time $t(S_1) + trans(M_1)$. As shown in figure 7.1 on the next page, if $t(S_1) + trans(M_1) \le t'(S_2)$ then the data arrives before the statement S_2 is reached, so there is no synchronization delay. On the other hand, if $t(S_1) + trans(M_1) > t'(S_2)$ then the processor that runs task T_2 will have to be idle until the data arrives, so the idle time is $t(S_1) + trans(M_1) - t'(S_2)$. Combining the two cases, the synchronization delay caused by the dependence δ is then $delay = \max\left\{0,\ t(S_1) + trans(M) - t'(S_2)\right\}$.

QED.

Conventional data dependence graphs [Wolfe89] have no provision for representing the concept of merging multiple data dependencies in a single synchronization point. Thus, we introduce here a new dependence relation called the *data dependence cluster*.

Definition 7.2. A *data dependence cluster*, Ψ, is a quadruple $(\Omega, \Delta, S_1, S_2)$ where Ω is a set of data dependencies from task T_1 to task T_2, Δ is the union of the data involved in the dependencies in Ω, S_1 the last statement in T_1 that must be executed before the data can be sent to T_2, and S_2 is the first statement in T_2 where the data involved in the dependencies in Ω must arrive or the execution of T_2 will be blocked.

The data dependence cluster is a generalization of a single data dependence. A data dependence δ involving the data d from statement S_1 to statement S_2 defines a data dependence cluster: $(\{\delta\}, \{d\}, S_1, S_2)$. And the effect of this data dependence cluster on the performance of the program is the same as that of the single data dependence.

Corollary 7.1.1 The data synchronization delay for a data dependence cluster that contains one data dependence is the same as the data synchronization delay caused by the dependence.

Many operations can be defined on the data dependence cluster, but here we will discuss only the union (called merge below) operation. We define the union of two data dependence clusters $(\Omega^a, \Delta^a, S_1^a, S_2^a)$ and $(\Omega^b, \Delta^b, S_1^b, S_2^b)$ to be $(\Omega^a \bigcup \Omega^b, \Delta^a \bigcup \Delta^b, S_1', S_2')$, where S_1' is S_1^a or S_1^b, depending on which statement occurs later lexicographically, and S_2' is S_2^a or S_2^b, depending on which statement occurs first lexicographically.

Definition 7.3. Two data dependence clusters $\Psi^a = (\Omega^a, \Delta^a, S_1^a, S_2^a)$ and $\Psi^b = (\Omega^b, \Delta^b, S_1^b, S_2^b)$ are called mergeable if there is no dependence δ from statement S_i to S_k such that S_i is between S_2^a and S_2^b in T_2 and S_k is between statements S_1^a and S_1^b in T_1.

Trying to merge two data dependence clusters that are not mergeable would violate the data dependence by moving the source of a data dependence beyond a statement that depends on it. Furthermore, this would cause the two tasks to deadlock since task T_1 would have to wait for data from T_2 before it could process the statement S_k, and task T_2 will have to wait for data from T_1 before it can process statement S_i.

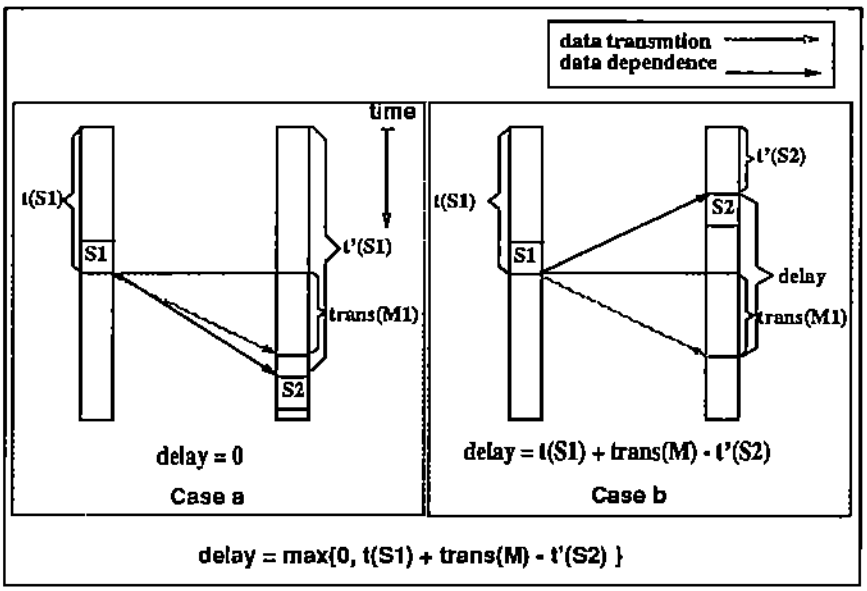

Figure 7.2. *Snchronization delay caused by inter-task dependency.*

The data dependence cluster can be used to guide the code generation for distributed memory architectures. For example, a write statement is generated after statement S_1 which sends the data in Δ and a read statement is generated in front of the statement S_2 to receive the data. Note that the data dependence clusters have mutually exclusive data dependence sets and form a partition of the data dependencies. A cluster $Cluster_1$ is defined to be '$<$' $Cluster_2$ if all statements in T_1 that are involved in the dependencies in the $Cluster_1$ are before those in $Cluster_2$. This defines a partial order of the clusters. Two clusters are said to be *adjacent* to each other if there are no clusters between them.

Lemma 7.2. Let δ^1 and δ^2 be two data dependencies from tasks T_1 to T_2, where δ^1 is from statements S_1 in T_1 to S_2 in T_2 and δ^2 is from statements S_3 in T_1 to S_4 in T_2. Let $t(S_i)$ be the execution time of the statements between the first statement in T_1 and S_i including that of S_i, and $t'(S_j)$ be the execution time of the statements between the first statement in T_2 and S_j, excluding that of S_j. Let M^i be the size of the data that causes the data dependence δ^i, and $trans(M^i)$ be the cost of transmitting the data of size M^i. Then, the delay in T_2 caused by the two data dependencies is the maximum of the two delays. In other words, the delay for task T_2 is:

$$\begin{aligned} delay &= \max\left\{ t(S_1) + trans(M^1) - t'(S_2),\ t(S_3) + trans(M^2) - t'(S_4) \right\} \\ &= \max\left\{ delay^1,\ delay^2 \right\} \end{aligned} \tag{7.2}$$

Proof:

By lemma 7.1, we know that the delay caused by dependence δ^1 is

$$delay^1 = \max\left\{ 0,\ t(S_1) + trans(M^1) - t'(S_2) \right\}$$

The problem may be divided into two cases based on the order of statements S_2 and S_4.

Case 1: $t'(S_2) \le t'(S_4)$ (as shown in figure 7.2):

Due to the dependence δ^1, a delay of $delay^1$ has to be inserted before the statement S_2; as a result, every statement in task T_2 after S_2 is delayed by this amount of time. So the statement S_4 will be reached at time $t'(S_4) + delay^1$, and by lemma 7.1, the delay caused by the dependence δ^2 becomes

$$delay^{2'} = \max\left\{ 0,\ t(S_3) + trans(M^2) - (t'(S_4) + delay^1) \right\}$$

$$= \max\left\{0,\ (t(S_3) + trans(M^2) - t'(S_4)) - delay^1\right\}$$

$$= \max\left\{0,\ delay^2 - delay^1\right\}$$

Therefore, the delay for both dependencies is given by

$$delay = delay^1 + delay^{2'} = delay^1 + \max\left\{0,\ delay^2 - delay^1\right\}$$

$$= \max\left\{delay^1,\ delay^2\right\}$$

Case 2: $t'(S_2) > t'(S_4)$ (as shown in figure 7.3):

Since the statement S_4 is in front of the statement S_2, $delay^2$ has to be inserted before statement S^4 delaying the time statement S_2 gets executed. So the new delay for statement S_2 is:

$$delay^{1'} = \max\left\{0,\ t(S_1) + trans(M^1) - (t'(S_2) + delay^2)\right\}$$

$$= \max\left\{0,\ (t(S_1) + trans(M^1) - (t'(S_2)) - delay^2\right\}$$

$$= \max\left\{0,\ delay^1 - delay^2\right\}.$$

This implies that the synchronization delay caused by dependencies δ^1 and δ^2 is

$$delay = delay^2 + delay^{1'} = delay^2 + \max\left\{0,\ delay^1 - delay^2\right\}$$

$$= \max\left\{delay^2,\ delay^1\right\}$$

This concludes the proof.

QED.

Note that the case b in figure 7.3 can occur on architectures that support alternative routing when the message size M^1 is very large. For other machines that use fixed routing between two processors, the first message will block the second message so this case is not possible. The lemma still holds because the transmission time for sending the second message will be much longer, thus causing a longer delay.

A direct generalization of the above lemma leads to the following lemma.

Lemma 7.3. When there is more than one data dependence between two tasks, the synchronization delay in task T_2 is determined by the dependence that causes the longest delay. That is, if there are n data dependencies from tasks T_1 to T_2 where the i-th dependence δ^i is from statements S_f^i to S_t^i, and the i-th delay, $delay^i$, is caused by δ^i, then the delay for task T_2 caused by the data dependencies from task T_1 is:

$$delay = \max_{i=1}^{n}\left\{t(S_f^i) + trans(M^i) - t'(S_t^i)\right\}$$

$$= \max_{i=1}^{n}\left\{delay^i\right\} \tag{7.3}$$

Proof:

The lemma can be proved by induction. The base case is when there are two data dependencies and is proven in lemma 7.2. Assuming that the lemma is true for any m data dependencies where $m < n$, we now proceed to prove for the case of n data dependencies. We also assume that the dependencies are ordered by

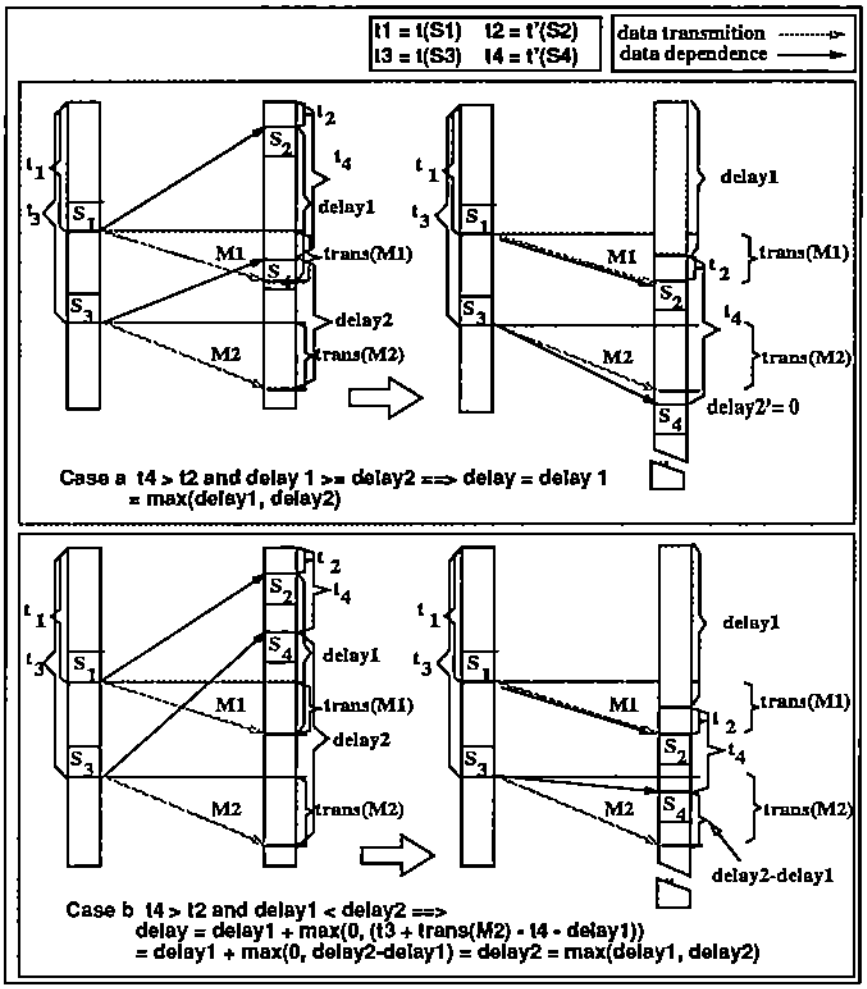

Figure 7.3. *The synchronization-delay caused by two data dependencies between tasks T_1 and T_2 (statement S_2 is before S_4).*

the order of the statements in task T_2. For the first $n-1$ data dependencies between the two tasks, assume that $delay^k$ is the largest delay in the delays caused by the dependencies; then by the assumption, the combined delay for the first $n-1$ dependencies is $delay^k$. Now consider the dependence δ^n, the statement S_i^n is delayed for $delay^k$ by the previous $n-1$ dependencies. The delay for the n-th dependence is then:

$$delay^{n'} = \max\left\{0,\ t(S_f^n) + trans(M^n) - (t'(S_i^n) + delay^k)\right\}$$

$$= \max\left\{0,\ delay^n - delay^k\right\}$$

So the overall delay of all n data dependencies is

$$delay = delay^k + \max\left\{0,\ delay^n - delay^k\right\}$$

$$= \max\left\{delay^k,\ delay^n\right\}$$

$$= \max\left\{\max_{i=0}^{n-1}\left\{delay^i\right\},\ delay^n\right\}$$

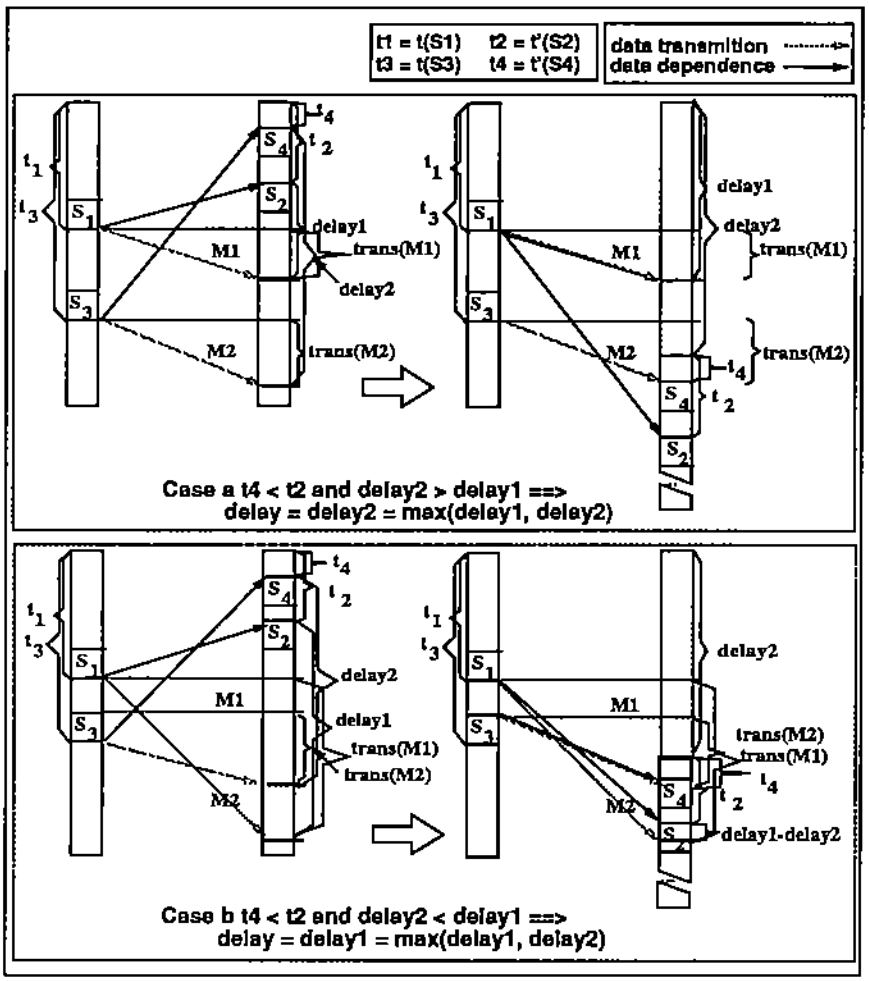

Figure 7.4. *The synchronization-delay caused by the two data dependencies between tasks T_1 and T_2 (statement S_4 is before S_2).*

$$= \max_{i=0}^{n} \left\{ delay^i \right\}$$

This completes the proof.

QED.

The following formula determines the new delay in task T_2 when the first message is merged into the second. The two cases that can occur are depicted in figure 7.4.

Lemma 7.4. If the two messages as defined in lemma 7.2 are merged into one, then the synchronization delay for task T_2 becomes:

$$delay = \max\left\{ 0,\ t(S_3) + trans(M^1 + M^2) - \min(t'(S_2),\ t'(S_4)) \right\} \tag{7.4}$$

Proof:

When the first message is merged into the second message, the data dependence from statement S_1 to statement S_2 is changed into a dependence from statement S_3 to statement S_2, and the size of the data to be sent from task T_1 to task T_2 is increased into M^1+M^2. At time $t(S_3) + trans(M^1 + M^2)$ the message will be available on task T_2 so the delay is this time minus the time the first statement involved in the data dependencies is reached which is $\min(t'(S_2), t'(S_4))$. This proves the formula.

QED.

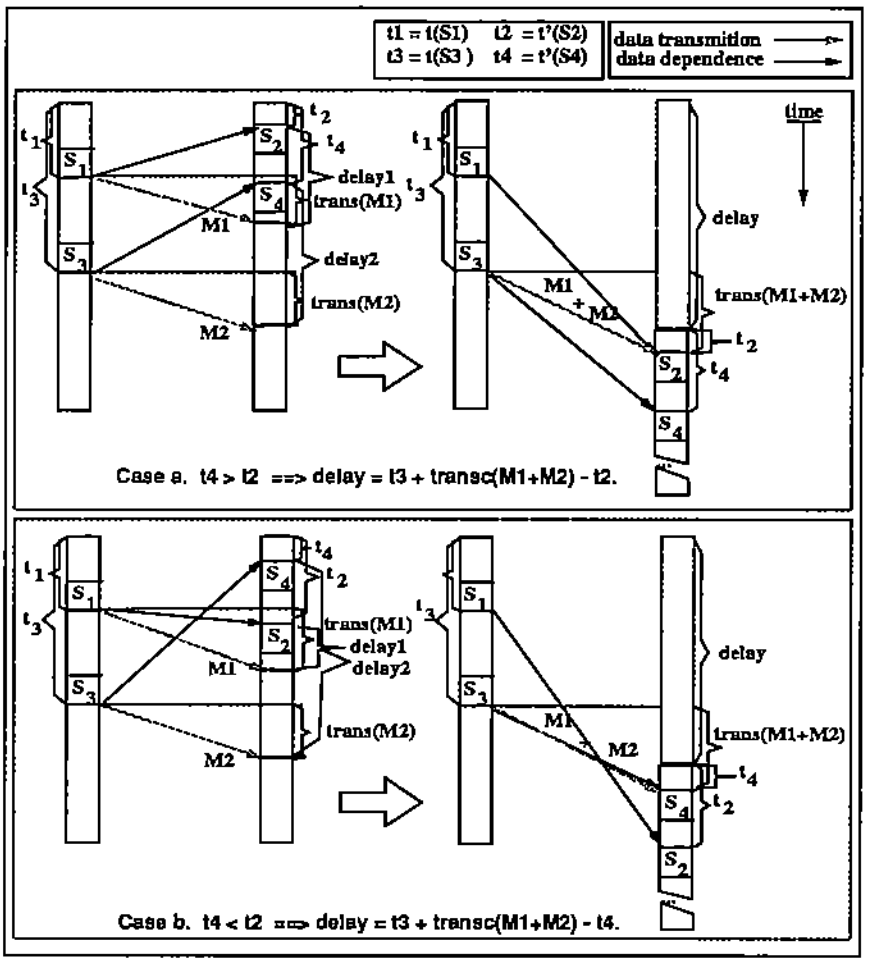

Figure 7.5. *The synchronization delay for merging two messages into one.*

The drawback of merging two messages is that the corresponding statements in task T_2 have to wait for the message to arrive, which might increase the data synchronization time.

On most distributed systems, long messages are preferred over short messages since the overhead for message startup is generally very high. The message transmission time *trans*() is given by:

$$trans(N, hops) = \alpha(hops) + \beta(hops) * N \tag{7.5}$$

where N is the size of the message, *hops* is the distance between the two processors, $\alpha(hops)$ is the message startup time and $\beta(hops)$ is the unit cost for data transmission.

Below we seek to determine the condition for which two mergeable messages can be profitably merged. Assuming that the same conditions exist as defined in lemma 7.2 and the two message are mergeable, let the delay between the two tasks after the two messages are merged, to be *delay*. We define C^m, the difference in delays before and after merging the messages, to be the difference between the new and the old delays. In other words, C^m is defined to be $delay - \max\{delay^1, delay^2\}$. The two messages can be profitably merged only if $C^m < 0$.

Note that if the original delay between the two tasks, $\max\{delay^1, delay^2\}$, is less than or equal to 0, that is there is no delay due to synchronization, then there is no point in merging the two messages.

The next lemma describes how C^m can be computed.

Lemma 7.5. If $\max(delay^1, delay^2) > 0$ then C^m is given by:

$$
\begin{cases}
t(S_3) - t(S_1) + \beta(hops) * M^2 & \text{if } t'(S_2) \le t'(S_4) \text{ and } delay^1 \ge delay^2 \\
t'(S_4) - t'(S_2) + \beta(hops) * M^1 & \text{if } t'(S_2) \le t'(S_4) \text{ and } delay^1 < delay^2 \\
t(S_3) - t(S_1) + t'(S_4) - t'(S_2) + \beta(hops) * M^1 & \text{if } t'(S_2) > t'(S_4) \text{ and } delay^2 \ge delay^2 \\
\beta(hops) * M^1 & \text{if } t'(S_2) > t'(S_4) \text{ and } delay^1 < delay^2
\end{cases} \tag{7.6}
$$

Proof:

By definition $C^m = delay - \max(delay^1, delay^2)$.

If $t'(S_4) \ge t'(S_2)$ and $delay^1 \ge delay^2$ (as in figure 7.2 case a) then the delay caused by the two data dependencies is $delay^1$. Since $delay^1 > 0$, the extra cost of merging the two messages is

$$
\begin{aligned}
C^m &= (t(S_3) + trans(M^1 + M^2) - t'(S_2)) - (t(S_1) + trans(M^1) - t'(S_2)) \\
&= (t(S_3) - t(S_1)) + (trans(M^1 + M^2) - trans(M^1)) \\
&= t(S_3) - t(S_1) + \beta(hops) * M^2.
\end{aligned}
$$

If $t'(S_4) \ge t'(S_2)$ and $delay^1 < delay^2$ (as in figure 7.2 case b) then the delay caused by the two data dependencies is $delay^2$. Since $delay^2 > 0$, the extra cost of merging the two messages is then

$$
\begin{aligned}
C^m &= (t(S_3) + trans(M^1 + M^2) - t'(S_2)) - (t(S_3) + trans(M^2) - t'(S_4)) \\
&= (t'(S_4) - t'(S_2)) + (trans(M^1 + M^2) - trans(M^2)) \\
&= t'(S_4) - t'(S_2) + \beta(hops) * M^1.
\end{aligned}
$$

If $t'(S_4) < t'(S_2)$ and $delay^1 \ge delay^2$ (as in figure 7.3 case a) then the delay caused by the two data dependencies is $delay^1$. Since $delay^1 > 0$, the extra cost of merging the two messages is

$$
\begin{aligned}
C^m &= (t(S_3) + trans(M^1 + M^2) - t'(S_4)) - (t(S_1) + trans(M^1) - t'(S_2)) \\
&= (t(S_3) - t(S_1)) - (t'(S_4) - t'(S_2)) + (trans(M^1 + M^2) - trans(M^1)) \\
&= (t(S_3) - t(S_1)) - (t'(S_4) - t'(S_2)) + \beta(hops) * M^2.
\end{aligned}
$$

If $t'(S_4) < t'(S_2)$ and $delay^1 < delay^2$ (as in figure 7.3 case b) then the delay caused by the two data dependencies is $delay^2$. Since $delay^2 > 0$, the extra cost of merging the two messages is

$$
\begin{aligned}
C^m &= (t(S_3) + trans(M^1 + M^2) - t'(S_4)) - (t(S_3) + trans(M^2) - t'(S_4)) \\
&= (trans(M^1 + M^2) - trans(M^2)) \\
&= \beta(hops) * M^1.
\end{aligned}
$$

QED.

Merging two messages into a single message has two distinct effects:

1. It increases the synchronization delay of the receiving processor that runs task T_2 as described in lemma 7.5 by delaying the sender.
2. It decreases the execution time of the processor running task T_1 by C^{send} (overhead of sending a message).

Although there is no clear-cut method for estimating the combined effects on the overall performance of the program, lemma 7.5 can be used as a guideline for deciding when to merge two messages. Messages should be merged only when the overhead for sending a message justifies the extra data synchronization cost caused by delaying the sending of the first data.

Heuristic 7.1. Assuming C^m is the overhead in task T_2 for merging the two messages and C^{send} is the overhead for sending a message then the messages can be merged when the following condition is satisfied:

$$
C^m = delay - \max(delay^1, delay^2) \le C^{send} \tag{7.7}
$$

where C^m is defined in lemma 7.5.

The above heuristic assumes that the delays in different tasks have the same effects on the overall performance of the program. This assumption is again conservative. Although changes in the execution time of any task would affect all tasks that interact with it, more than likely the overhead in task T_2 can be masked by overlapping the communication with computation. On the other hand, the saving by eliminating one message communication
will directly decrease the execution time of T_1.

Based on heuristic 7.1 and lemma 7.5, we can derive a heuristic-guided algorithm for deciding when messages can be profitably merged. The problem of minimizing data synchronization can be defined as a problem of partitioning the data dependencies into data dependence clusters. Initially, we assume that each data dependence forms a data dependence cluster by itself. We then proceed to merge (union) the clusters into larger clusters until the merge is no longer beneficial. This algorithm is applied to each pair of tasks $\{T_1, T_2\}$ that have cross-task dependencies between them.

Algorithm 7.1. Message merging.

For each pair of tasks T_1 and T_2 with data dependence from T_1 to T_2 do

1. Initialize each data dependence to form a data dependence cluster of its own.
2. For each data dependence cluster pair (Ψ^i, Ψ^j) do

   If Ψ^i and Ψ^j are mergeable then

   Calculate C^m, the cost of merging Ψ^i and its adjacent data dependence cluster Ψ^j, and set $B^{i,j} = C^{send} - C^m$

   else

   Set $B^{i,j}$ to be $-\infty$.

   end if

   end for
3. Sort the pair of data dependence clusters (Ψ^i, Ψ^j) in a heap based on $B^{i,j}$
4. While the minimum B over all pairs of data dependence clusters is positive do

   Merge Ψ^i and Ψ^j into $\Psi^{i'}$

   Recompute the value $B^{i',j}$ for all Ψ^j and adjust the heap.

   end while.

END.

This algorithm has the complexity of $O(n^2 * \log(n))$ where n is the number of cross-task data dependencies. This is because in step 2 the cost calculation was repeated $(n-1)^2$ times. In step 3, the cost of sorting n^2 numbers is $O(n^2 * \log(n))$. There will be at most $n-1$ merges in step 4. So the overall complexity is $O(n^2 * \log(n))$.

One can improve the above algorithm by applying a heuristic that merges only adjacent data dependence clusters. This heuristic makes good sense, because the synchronization delay in T_2 is directly proportional to the distance between statements S_1^a and S_1^b.

Algorithm 7.2. Message merging (merge only adjacent dependencies).

For each pair of tasks T_1 and T_2 that have data dependence from T_1 to T_2 do

1. Initialize each data dependence to form a data dependence cluster of its own.
2. For each data dependence cluster Ψ^i do

   Calculate C^m, the cost of merging Ψ^i and its adjacent data dependence cluster Ψ^{i+1} (if they are mergeable) and set $B^i = C^{send} - C^m$

   end for
3. Sort the data dependence clusters in a heap based on B^i
4. While the minimum B over all data dependence clusters is positive do

   Merge Ψ^i and its adjacent data dependence cluster Ψ^{i+1} into $\Psi^{i'}$

   Compute the value B' for $\Psi^{i'}$ and adjust the heap.

end while.

END.

Lemma 7.6. The complexity of the algorithm 7.2 is $O(n * \log(n))$, where n is the number of cross-task dependencies.

Proof

Initially there are n dependence clusters, and after each merge there is one less data dependence cluster. So in the worst case, the algorithm can execute the while loop at step 4 at most $n-1$ times. In step 3 sorting n numbers takes $O(n * \log(n))$. Thus the overall complexity of the algorithm is $(O(n * \log(n))$.

QED.

Note that if we omit the sorting in step 3 in algorithm 7.2 and merge the data dependence clusters Ψ^i and Ψ^{i+1} if B^i is positive, then, the algorithm 7.2 is linear. The drawback of this heuristics is that we lose the optimality claim of algorithm 7.1 when speeding up the algorithm.

When loops are parallelized, it is usually done by blocking the loops into a parallel loop and a sequential inner loop. The non-local data references inside the sequential loop often define a very regular pattern of accesses. Thus the message consolidation algorithm can take advantage of this regularity and does not need to unroll the loop to consolidate the messages. Instead, the algorithm can work on the statements inside the loop as well as statements at the end of the statement blocks by assuming that the loop wraps around once. The results can then be generalized to all loop instances. This implies that the complexity of the algorithm for parallel loops is the number of cross-task data dependencies in the inner-loops.

Statement reordering can be used to move the definition of the data that is the source of a cross-task dependence to as early in the code as possible and move the use of the cross-task data to as late as possible. This has the effect of minimizing the synchronization delay.

The transformation *array reshaping* that we discussed in section 7.3 can be used to condense the size of the data to be moved across the network to further decrease the data transmission time. The dependence graph is adjusted so that the data references in the receiving task depend on the local variables that hold the arriving messages instead of the original variables. This keeps the dependence graph in a consistent state.

7.4.3. Summary

The major problems in parallelizing sequential programs for distributed memory parallel computers lie in distributing data into local memories and converting data dependencies into messages. In this paper, we have discussed the problem of reducing cost of data synchronization by consolidating data dependencies and thus the corresponding messages. We introduced a special kind of data dependence called the data dependence cluster. The data dependence clusters represent the generalized data dependence relations for programs under the message-passing paradigm. We analyzed performance of merging data dependence clusters and messages based on the estimated parallel execution time of the program. The message consolidation techniques introduced in this paper can reduce the communication overhead for parallel computers significantly when they are applied appropriately. The algorithm for message consolidation that we have presented here utilizes a heuristic-guided performance prediction model (see chapter 6) to decide whether two messages can be beneficially merged. The algorithm is optimal in the sense that it minimizes the performance predicted by the model.

7.5. Algorithm Substitution and Fine-Tuning with Pre-Optimized Algorithms

Some widely used numerical algorithms have been explored extensively on different multiprocessor systems. The optimizations of these algorithms often involve fundamental algorithm changes in order to utilize the special parallelism features provided by the target machines. Some of the techniques and heuristics of modifying these algorithms are problem-dependent and are applicable only for some particular problems and architectures. The importance of these algorithms makes optimizing them important, but one does not want to add the specific heuristics and techniques used in optimizing them into the knowledge base unless they can be used by other problems. One solution to this problem is to include the pre-optimized algorithms for these kinds of problems in the knowledge base of the system. When a program matches the pattern of a pre-optimized algorithm, rather than performing a series of transformations, the algorithm of the program is replaced by the pre-optimized version.

Some of the pre-optimized algorithms can actually be achieved by a series of basic transformations. In these cases, the use of transformations or pre-optimized algorithms is debatable. The pre-optimized algorithms eliminate the lengthy intermediate transformations. Once the pattern is recognized, the system can directly substitute the parameters of the algorithm to translate the program into the desired forms. Also, pre-optimized algorithms can achieve optimal results for the particular problems that might well be beyond the ability of the transformation system. Using pre-optimized algorithms helps to cut down the size of the rules applied to general problems, because those heuristics that are only applicable to the special pre-optimized algorithms need not be included. On the other hand, the pre-optimized algorithms increase the size of the knowledge base, and the pattern recognition test for a pre-optimized algorithm adds penalties to all problems that may not be related to the algorithm at all.

The choice between fine grain heuristic-directed transformations and the pattern-directed pre-optimized algorithms relies solely on the state of the art decisions of the system engineers. Our approach is that heuristics that are only applicable to very limited classes of applications are not included, and pre-optimized algorithms are only used for problems that cannot be optimized by the general heuristics and transformations.

Due to the nature of this approach, we will explain the ideas through an example. The example we consider here is the array accumulation problem. This is an interesting problem because the program is simple but it contains data dependencies and memory accesses that may serialize the computation. This example allows us to illustrate the heuristics of selecting and fine-tuning the pre-optimized algorithms as well as their applicability. It also shows the importance of the resolution of memory contentions. The following is the general form of the array accumulation problems.

```
var   a : array [1..B] of integer;
for i in 1 .. N do
      a[f(i)] += g(i);
end for
```

In the above program, f is a function that maps the range of the loop index to the range of index of array a. For the sake of simplicity, we assume that the accumulation statement is enclosed in a single loop. In more general cases, the accumulation statement may be nested in multiple loops or the array may be a multi-dimensional array.

If the index function f is a one-to-one function, then f is a permutation on a subset of interval $[1..B]$. In this case, values of $g(i)$ are accumulated into distinct elements of array a. When this program is run on a multiprocessor system, no two processors will update the same memory cell at the same time. However, memory/bus contention problems may still exist because of the limit of bus or network bandwidth, but memory locks are not needed.

For machines with P processing units, one trivial approach is to divide the program into P equal sized tasks and run them on the P processors as shown below.

```
var   a : array [1..B] of integer;
forall i in 1 .. P do
      var s : integer;
      s := N / P;
      for j in (i-1) * s .. i * s - 1  do
            a[f(i)] += g(i);
      end for
end forall;
```

For this approach, no pre-optimized algorithm is needed but the speed-up may not be impressive because the efficiency of the approach depends on the values of the index function f, the memory and bus bandwidths as well as the degree of memory interleaving. If more information about the index function is available, then better optimizations are possible.

For example, if the index function is a linear function with respect to the loop indices and the right hand side of the assignment takes about the same amount of time to be processed for all loop instances, then the memory updates may be regulated by updating the memory according to the memory interleaving. This can be accomplished by "index shifting" to shift the index function such that the memory update requests are evenly distributed to all the memory modules. This approach might be the fastest that we can achieve on some

particular machines since the memory updates are processed at their maximum extent.

If f is a constant function then the values of $g(i)$ are accumulated at the same memory cell $a[C]$, where $C = f(i)$. In this case, the problem is called a one bin accumulation problem. If this program is to be executed on a multiprocessor computer with P processors, each instance of the loop will have to compete to update the same memory cell $a[C]$. On machines with a combining network that supports the "fetch and add" operations such as the NYU Ultracomputer [Schw80], the accumulation requests can be combined in the network as the requests are routed to the memory. When the requests arrive at the memory module, there will be only one combined memory update request remaining; thus no memory lock is needed. The function call *barrier()* synchronizes the processors and guarantees that each processor starts the next operation at the same time which is essential for the coordination of the "fetch and add" operations.

```
var   a : array [1..B] of integer;
forall i in 1 .. P do
      var   s: integer;
            index, value: integer;
      s := N / P;
      for j in (i-1) * s .. i * s - 1  do
            index = f(i);
            value = g(i);
            barrier();
            fetch_and_add(a[index], value);
      end for
end forall
```

For machines without a combining network and a "fetch-and-add" operation capability, mutually exclusive accesses to $a[C]$ need to be enforced to avoid memory update conflicts. This might serialize the memory updates and lose all the parallelism of the machine. One possible solution is to block the loop into P chunks and allocate them to P processors; each processor accumulates the values of $g(i)$ in a local counter. These counters are summed up through a tree-sum algorithm after all processors finish their local accumulations. Only one synchronization is needed before the tree-sum operation is started.

```
var   a : array[1..B] of integer;
      private : array [1..P] of integer;
forall i in 1 .. P do
      var   s : integer;
      s := N / P;
      private[i] := 0;
      for j in (i-1) * s .. i * s - 1 do
            private[i] += g(j);
     end for
end forall
barrier();
a[f(1)] := tree_sum(private[*]);
```

If the expression $g(j)$ contains no function calls then the private accumulations can be computed in N/P cycles and the tree-sum operation can be computed in $O(logP)$ cycles.

If the function f is neither a constant function nor a one-to-one function, this problem is called a *multiple bins accumulation problem* because the values of $g(i)$ are accumulated into many elements of array a simultaneously. A significant number of synchronizations are needed because of the unpredictability of the values of the function $f()$. Processors may compete to update any of the array elements at any instance, so memory locks need to be placed on all elements of the array a to guard correct memory updates.

All these three kinds of array accumulation problems are common in practical applications. One particularly interesting example of these is the image processing algorithm called "histogramming." The histogramming problem is a special case of the multiple bins accumulation problem whose index function is array reference. In image processing, a picture frame is represented by a two dimensional array of points called *pixels*. Each *pixel* has a small value between $0..(b-1)$ (typically an 8-bit number) that represents the grey scale value

or the color RGB value of the point. The histogramming involves keeping track of the occurrence of each grey scale value in the picture.

Throughout this section, the multiple bins accumulation problem is used to demonstrate the general ideas, but at times when more detailed illustration is needed the histogramming problem will be used.

The multiple bins problem can be detected by the system by matching the program with the general form of the multiple bin described above. If the index function *f* involves only loops indices, then the system can detect the type of the problem at hand by applying array subscript tests. However, when there are variables other than the loop indices involved, determining whether the index function is a constant function or a one-to-one function may not be trivial for the system. For example, the index may be an array reference whose *i*-th element has value *i*. Although the user may know that the index function is an identity function, the system will not be able to determine this fact simply by the static analysis. In these cases, the system will have to rely on the user interaction to provide help. If the subscript test fails and the user cannot provide help, the more complicated multiple bin problem is assumed.

When a multiple bins accumulation problem is recognized by the system, pre-optimized algorithms for this problem are considered. When there is no pre-optimized version for the target architecture, the computational model needs to be analyzed to see if any pre-optimized version can be applied to the target architecture. Heuristics are also considered in fine-tuning the pre-optimized algorithm to match with the target machine.

For machines that support combining networks and "fetch and add" operations, the accumulation operations in the multiple bins accumulation problem can be translated into "fetch and add" operations. If we divide the program into P tasks by loop blocking and run the tasks by *P* processors, there will be at most *P* "fetch and add" requests for the same memory location at each cycle. Requests that have the same destinations are combined in the network and merged into one request when arriving at the memory cell. The "fetch and add" operations eliminate the need for memory locks so the overall performance of the algorithm will be improved. However, the speed-up of this approach may not be significant, because there may be many memory cells to reference in the same operation cycle. Depending on the reference patterns, the memory update requests may pile up in the network and memory modules may thus cause network saturation.

The two-phase private counters approach we used in the one-bin accumulation problem can be extended to solve the multiple bins accumulation problem on multiprocessor machines that have several memory modules. During the first phase, an array of private accumulation counters is created for each processor that executes the program. Each processor gets a share of the job and updates the private counters independently. In the second phase, these private counters are summed up either by tree-sum or other available parallel summation algorithms.

The memory updates of *a*[*f* (*indices*)] in the original program are irregular and unpredictable. In the first phase of the two-phase approach, the memory access pattern of *private_a* is unpredictable. But, since each processor exclusively owns its private counters, the private counters can be updated simultaneously and independently. One possible problem is that when the machine has a small cache, the irregular updates of the array private_a may produce a high cache miss ratio. However, in most problems, the private counters are fairly small. For example, the private counter array in the histogramming problem is of size 2**b*. When *b* is equal to 8, the size of the private counter array is 256 so the entire array can reside in the cache throughout phase 1.

On the other hand, the memory update pattern in phase 2 is very regular. Very few synchronizations are needed. Processors can cooperate to sum up the private counters. On most machines, the pre-defined tree-sum algorithm will provide a reasonably good speed-up.

Note that this approach of introducing private counters and dividing the memory accesses into two phases is based on the expertise about this particular array accumulation problem and may not be applicable to other problems. Therefore, the pre-optimized algorithm is used. The pre-optimized algorithm did not specify where the private accumulation counters should be allocated because this problem falls into the category of the array allocation problem and the general heuristics about array allocation can be used to decide how the private counters should be allocated. In general, not all the details in pre-optimized algorithms need to be implemented because some of them are covered by general heuristics. As a result, some of the variations of the pre-optimized algorithm can be left undetermined until the actual program is substituted. The unspecified part can then be obtained by applying the general transformations.

```
var   a : array[1..B] of int
      private_a: array [1..P, 1..B] of integer;
      s : integer;
s := N / P;
forall i in 1 .. P do
      for j in (i-1) * s .. i * s - 1  do
            private_a[i, f(j)] += g(j);
      end for;
end forall;

barrier();
for j in 1 .. B do
      a[j] := tree_sum(private_a[*, j]);
end for;
```

This approach should cut down the number of pre-optimized algorithms that the system needs to store. The implementation of the function *tree_sum*() may vary from machine to machine. For machines that have combined networks, the summations can be accomplished by merging the "fetch and add" requests in the network. In this case, the last loop that performs the *tree−sum* can be translated into:

```
forall i in 1 .. P
      for j in 1 .. B do
            fetch_and_add(a[j], private_a[i, j]);
      end for;
end forall;
```

At each cycle, the P processes will generate P "fetch and add" requests to the same memory location. These requests are combined in the network and merged into one request when arriving at the memory. Since each processor accesses the same memory location at the same cycle, no hot spot network saturation problem will ever occur. This last statement will be valid only when all the processors are homogeneous and the system provides a barrier synchronization routine that can start all processors at the same time

On other machines that do not support "fetch and add" operations, the tree-sum operations can be handled by simulating the binary tree network. We can parallelize the outermost loop by synchronizing the memory accesses of the tree-sum operations.

```
for pid in P .. 1 step -1 do
   local private_a: array [1..P, 1..B] of integer;
   for i in 1 .. B do
      if (pid == 1) then
         a[i] := private_a[pid,i] + private_a[pid*2,i] + private_a[pid*2+1,i];
      else if (pid <= P / 2) then      /* is not a leaf */
         private_a[pid,i] += private_a[pid*2,i] + private_a[pid*2+1,i];
      end if
   end for
end for
```

In the above program, there are two pairs of dependencies from *private_a*[*pid**2,*i*] and *private_a*[*pid**2+1,*i*] to *private_a*[*pid*,*i*]. These dependencies correspond to the inherent characteristic of the tree operations, and the operations in the intermediate nodes of the tree cannot be executed until the children of the node finish the computations. If the array *private_a* is allocated in the shared memory, then semaphores need to be inserted to enforce the order of the tree computations.

```
var     syn: array[1..P] of semaphore;
forall pid in P .. 1 step -1 do
   local private_a: array [1..P, 1..B] of integer;
   for i in 1 .. B do
      if (pid == 1) then
        a[i] := private_a[pid,i] + private_a[pid*2,i] + private_a[pid*2+1,i];
      else
         if (pid <= P / 2) then     /* is not a leaf */
            wait(syn[pid*2]);   wait(syn[pid*2+1]);
            private_a[pid,i] += private_a[pid*2,i] + private_a[pid*2+1,i];
         end if;
         signal(syn[pid/2]);
      end if;
   end for;
end forall;
```

Here *signal* () and *wait* () are primitives for synchronization. If the array *private_a* is in the local memory, then the values can be passed to other processors by either copying them to shared memory with synchronization locks or sending them through the inter-processor communication channels.

For a non-shared memory machine like Pringle, the "tree-sum" operations are pipelined through a tree configuration of the machine. The switches in the network are configured such that the processor with processor id pid is connected to its parent whose id is $pid/2$ and its two children whose processor ids are $pid*2$ and $pid*2+1$. The processor with processor id 1 is the root of the tree. The synchronization and the buffers are hidden in the implementation of the channel variables on the machine.

The pipelined *tree–sum* algorithm can be executed in time $C * \log P * B$ where C is a small constant.

```
for i in 1 .. B do
      if (is_leaf(pid)) then
            CH_parent <- private_a[pid, i];
      else
            l <- CH_l_child;  r <- CH_r_child
            CH_parent <- l + r + private_a[pid, i];
      end if;
end for;
```

The unpredictable memory access pattern in these kinds of programs causes loop optimization techniques to be ineffective no matter how the control structure is modified. In this example, creating private accumulation counters regulates the memory update patterns and eliminates the need for memory locking. The tradeoff is that more memory cells and computing cycles are used to store and sum up the private counters. These costs are constant with respect to the number of processors P and the size, B, of the array a. Also, these costs may be compensated for by minimizing synchronization and resolving memory contentions.

For problems like the histogramming program, the memory accesses pattern is highly data dependent. There is no easy way for the compiler to tell whether the extra cost of manipulating the private bins justifies the synchronization costs it saves. Heuristics are used in deciding whether pre-optimized algorithms are suitable and beneficial. Use of the heuristics also allows the compiler to be more aggressive in parallelizing the program.

7.5.1. Summary

In this section, the use of pre-optimized algorithms is demonstrated; variations of the algorithm are used for different architecture configurations. The choice of the variations in implementation is determined by a set of rules based on the computational model. After the algorithm substitution, basic transformations are applied to match the algorithm with the computational model better.

A pre-optimized algorithm may use other pre-defined algorithms. For example, the two-phase approach in solving the multiple-bin accumulation problem uses the pre-optimized pipelined tree-sum algorithm. The actual implementation of the tree-sum operation is based on the computational model.

The pre-optimized algorithms differ from the fine grain heuristics in the abstract level of the knowledge: the pre-optimized algorithms are pre-packed special purpose heuristics which are only suitable for the particular problem they are designed for. Better optimizations may be achieved because the pre-optimized code has been optimized extensively for the particular target machine. The fine-tuning process we described above can be used to utilize previously optimized code even though the target machine may not completely match the architecture for which the program is optimized.

CHAPTER 8
IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter we describe an implementation of an intelligent parallel programming environment designed to demonstrate the ideas we have proposed in this thesis. Some experimental results are also presented.

8.1. An Experimental Intelligent Parallel Programming Environment

In this experiment, we constructed a prototype intelligent parallel programming environment, called *IntelliCompiler*. IntelliCompiler incorporates some new concepts and distinct features:

1. No pre-selected program transformation sequences are assumed; rather, the program optimization unit analyzes features of the program and the target machine and utilizes heuristics in the knowledge base to choose the program transformation sequences dynamically.
2. When there are factors that cannot be resolved by the static analysis, multiple paths of control flow are evaluated. If differences in performance are significant among different control flows, run-time tests are generated to decide the best control flow at run-time.
3. The systems provide supports for representing features and knowledge of the architectures and programs explicitly.
4. A performance prediction unit is incorporated in the decision-making process to estimate the effects of the transformation.
5. The system supports various interaction and optimization degrees and is designed to incorporate self-learning modules and to accommodate different architectures.

8.1.1. The System Architecture

The major components of the parallel programming environment include:

- Multiple front-ends for different programming languages.
- Multiple back-ends.
- An intelligent program optimization system.
- A machine knowledge manipulation system.
- A user interface.

8.1.1.1. The Front-Ends and the Back-Ends to the Programming Environment

Currently, front-ends to the system include parsers and program dependence graph generators for Blaze, C, and Fortran†. All front-ends generate BLAZE program dependence graphs as the input to the intelligent program optimization system; this allows the latter to be language independent. The BLAZE program dependence graphs are generated by the program dependence graph generator in the front-ends‡.

The back-ends of the system include unparsers that generate high-level programming languages for Blaze, E-Blaze, C, Fortran, and EPEX-C programs§ and code generators for Sequents and Suns.

---

† The Blaze front-end was built with contributions from the following individuals: P. Mehrotra, K. Wang, C. Koelbel, G. Shannon, and K. Miller. The C and Fortran front-ends were developed by D. Gannon and his students at Indiana University.

‡ The Blaze program dependence graph generator is written by K. Wang and C. Koelbel.

§ The Blaze and E-Blaze unparser were written by K. Wang. The C and Fortran unparser were written by D. Gannon

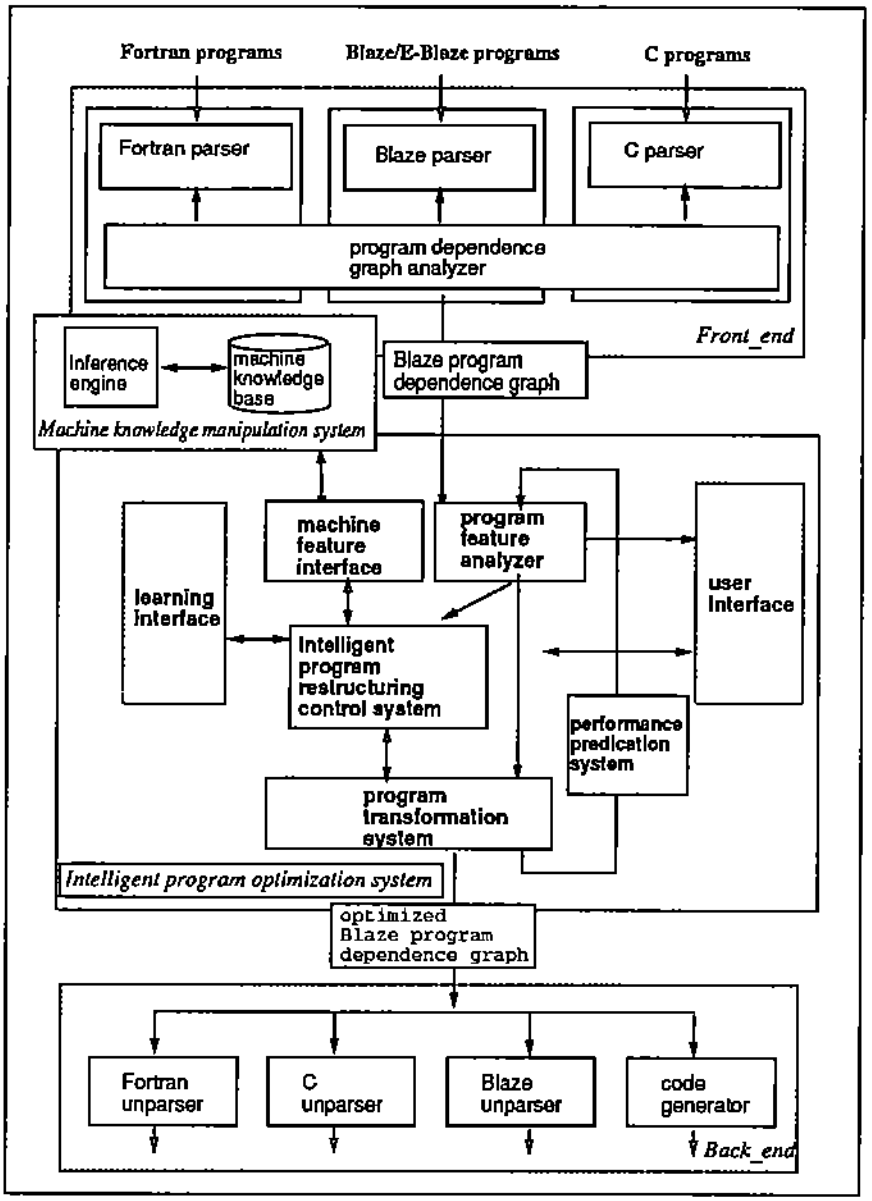

Figure 8.1. *The structure of the parallel programming environment.*

8.1.1.2. The Machine Knowledge Manipulation System

The machine knowledge manipulation system, as discussed in chapter 5, uses an object-oriented knowledge representation scheme and is equipped with an inference engine to perform reasoning on the features of the machines. Not all features of the machine need to be specified by the system programmer. Based on some basic features, the system runs a test program to collect some machine-dependent features (such as the size and magnitude of floating points, unit time for floating point operations, and memory reference costs) or language-dependent features (such as overheads for loops, array address calculations, and procedure calls). Some features may be derived by the inference engine of the system based on the existing knowledge of similar architectures. The machine knowledge manipulation system supports three different modes:

---

and his students. A different version of the C unparser for nCUBE and iPSC was written by C. Koelbel and P. Mehrotra. The EPEX-C unparser was written by K. Wang and C. Koelbel while they were supported as summer students at the IBM T. J. Watson Research Center, 1986.

- *Query mode.* This is the usual interface to the programming environment. Given a name (or alias) of a target machine, the system returns a list of the features currently know about the machine.
- *Knowledge update mode.* This starts an interactive session to install new machines, new features of a known machine, or modify the existing knowledge.
- *Inference mode.* This starts an interactive session with an interface to the SQL database. Its reasoning capability allows the user to compare relations between features of the machines.

8.1.1.3. The Intelligent Program Optimization System

The structure of the intelligent program optimization system includes a machine feature interface, a program feature analyzer, a program transformation system, an intelligent program restructuring control system, a learning interface module, and a user interface module. A figure of the program optimization system is shown in figure 8.2.

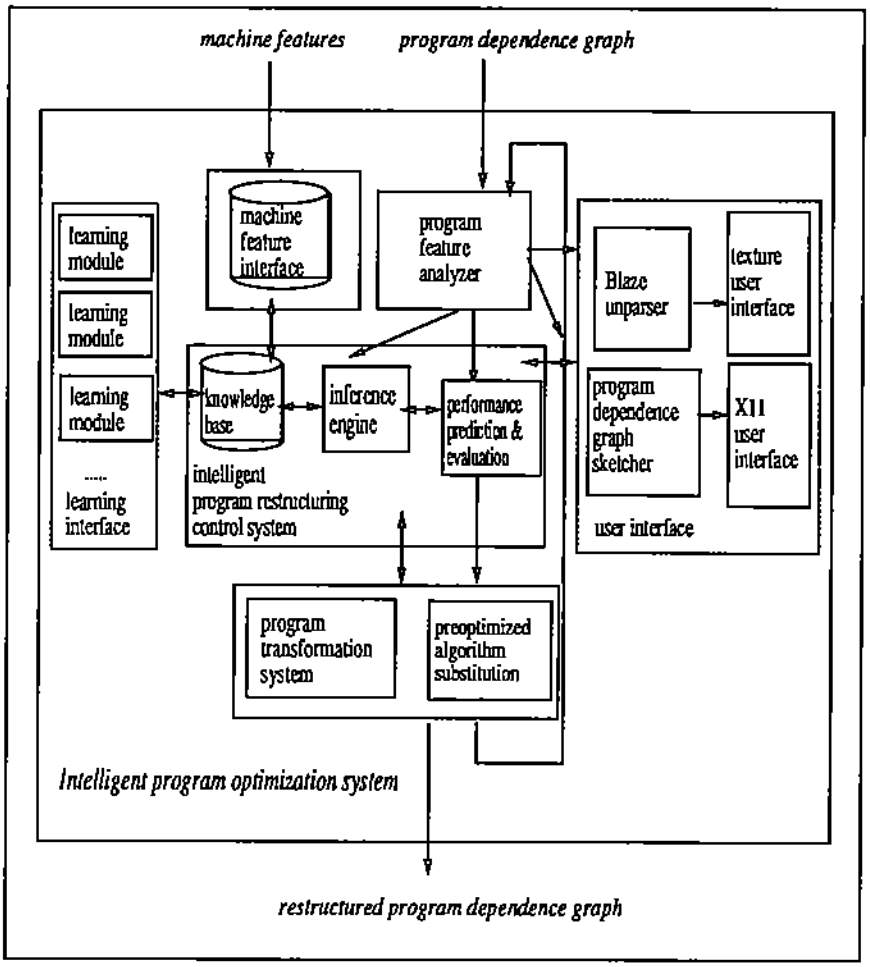

Figure 8.2. *The structure of the intelligent program optimization system.*

The machine feature interface stores the information generated by the machine knowledge manipulation system. It also acts as the interface between the program optimization system and the machine knowledge manipulation system. The system can use the interface to query the machine knowledge manipulation system for properties or relationships of machine features.

The program feature analyzer abstracts the features of the program based on the program dependence graph. Examples of such information include whether the program fragment matches a known algorithm (that has been previously optimized), which arrays are most critical from the point of optimization, whether the program has cross task dependence, etc.

The program transformation system contains various program transformation techniques. The transformations are organized into groups based on the objectives of the program-restructuring process (as discussed in chapter 4). Each program transformation is a module that consists of the knowledge and procedures for testing the applicability of the transformation, detecting opportunities for applying the transformation, evaluating effects of the transformation on the program, deciding the parameters of the transformation, and carrying out the transformation (by modifying the program dependence graph).

Table 8.1 shows the list of program transformations that are implemented and have heuristics for utilizing them encoded in the knowledge base of the system.

Table 8.1. *List of the program transformations that are currently implemented in the system.*

Name	*GO*	*VC*	*TC*	*PA*	*MU*	*AT*	*MS*
Statement reordering	•	•				•	•
Statement splitting		•				•	•
Forward substitution	•						•
Loop blocking	•	•	•				
Loop interchanging	•	•	•	•	•	•	
Loop merging	•		•			•	•
Loop distribution	•	•	•			•	
Loop unrolling	•	•				•	•
Index shifting		•				•	
Vectorization		•					
Cycle shrinking			•				
Array block transfer					•		
Array copying					•	•	
Array reshaping					•		•
Array localization					•		
Message consolidation							•
Run-time scheduling			•	•			
Do-across scheduling			•	•			•

The abbreviations used in the above figure are explained in the following table.

Abbrev.	*Subgoal*	*Abbrev.*	*Subgoal*
ET	*Enable other Transformation*	*GO*	*General Optimization*
MS	*Minimizing Synchronization*	*TC*	*Task Creation*
MU	*Memory Utilization*	*VC*	*Vectorization*
PA	*Processor Allocation*		

8.1.1.4. The Intelligent Program Restructuring Control System

The intelligent program restructuring control system utilizes the feature-directed program optimization paradigm and the hier-blackboard architecture. Its duties include choosing the appropriate optimization focuses, selecting and carrying out applicable program transformations, giving necessary explanation, and evaluating the result.

The intelligent program restructuring control system consists of an inference engine, a knowledge base, and a performance prediction subsystem. An explanation mechanism and help utility is also supported for users who choose to interact with the system. The inference engine is based on the blackboard architecture and features opportunistic reasoning. The knowledge base contains heuristic-oriented rules for guiding the program transformation system during the program restructuring process. The rules are organized into the knowledge sources based on the structure discussed in chapter 4. The performance prediction subsystem contains a set of evaluation functions which can be dynamically integrated to estimate the performance of the

current program on the target machine to aid the decision-making process. Different sets of evaluation functions can be used in different stages and parts of the program optimization process based on the available resources and the optimization degree. The details of the performance prediction system are described in chapter 6.

8.1.1.5. The User Interface

The user interface module contains a Blaze/Kali unparser, a textural user interface, an X Window user interface, and a program dependence graph sketcher. The dependence graph sketcher and the Blaze unparser are used to show intermediate states of the optimized program in graphic and textural forms. The graphic sketcher can display the program dependence graph on workstations that support the X Window system or print hard copies of the program dependence graphs. The unparser translates the program dependence graph of the program being optimized into user-understandable Blaze-like code. Both tools are intended for users who want to have a high degree of interaction to control the program restructuring process. It also helps the user to understand the reasons behind the decision-making process of the system.

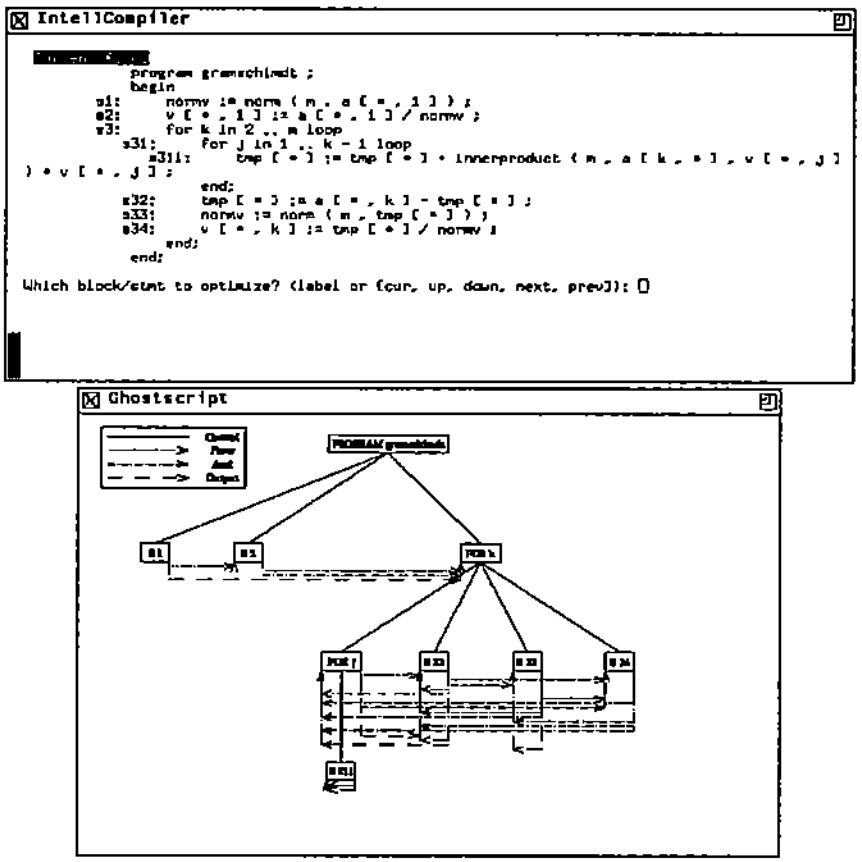

Figure 8.3. *The text and graphic form of the dependence graph generated by the Blaze unparser and the Blaze dependence graph sketcher.*

The system uses the UNIX socket-primitives to communicate with the Prolog process which is running in the background. The communication between the program optimization system and the front-end is through UNIX files in the form of BLAZE program dependence graphs.

The text user-interface is based on a Prolog library that we developed for building a general text based user-interface for Prolog programs. The Prolog library supports cursor movements and screen updates, dynamic menu-action selection mechanism, explanation and help utilities. The cursor movements and screen updates allow the menu selection to be used on dumb terminals. It provides full support for utilizing the terminal capability by consulting the terminal capability database. The dynamic menu (with actions associated with the entries) is a versatile text-driven menu mechanism for interactive control of the system. Items or actions

of the menu can be modified with other built-in menus or created at run time. A caching scheme in the ExpShell (described below) supports dynamic loading of the menu so that menus can be loaded at run time dynamically.

8.1.1.6. The Structure of the System

The entire program restructuring system is implemented on top of a hierarchy of expert system tools. These tools include a hier-blackboard system, an expert system shell called ExpShell, and a C-Prolog interpreter which runs on the UNIX† operating system. The architecture of the underlying system is shown in the following figure.

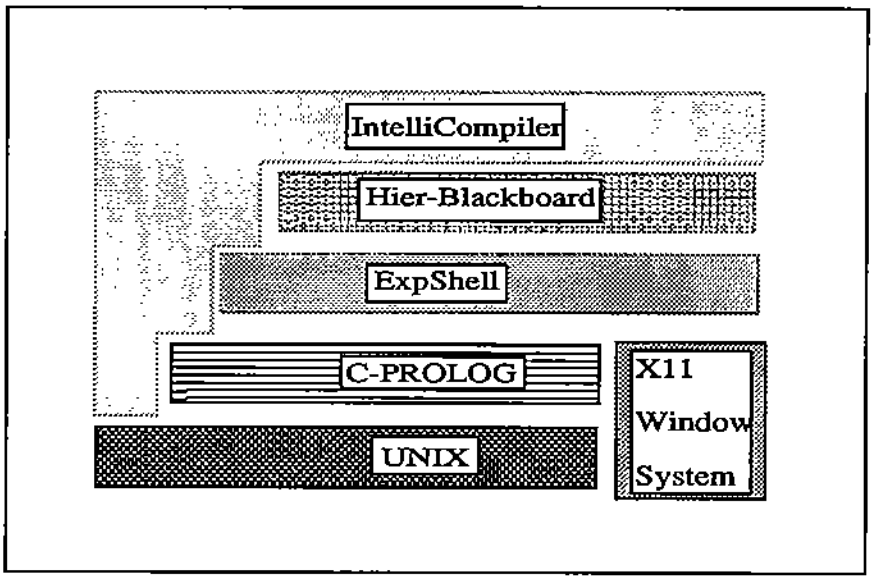

Figure 8.4. *The architecture of the underlying systems for the program restructuring system.*

The hier-blackboard system was described in detail in chapter 4.

The ExpShell was built for the implementation of the intelligent programming environment, but it was carefully designed so that it can be used to construct other expert systems by simply supplying it with appropriate domain knowledge. The ExpShell library contains utilities for supporting unification and inference, list manipulation, interface to the UNIX operation system, input/output and cursor movement control, dynamic menu manipulation, user interface, knowledge manipulation and installation, rule caching, and Prolog program analysis and debugging. The system debugging tool can discover errors where undefined functions are called. It also warns about the following potential problems: finding a vector that appears only once in a predicate (possibly due to a misspelled name), two functions with the same name (same or different number of arguments) but which have other functions defined in between (misspelled name or poor choice of names), a function whose predicates appear in two or more files, and asserting a function that is also defined in a file. The debugger also builds up a cross reference map of the predicates, so that optimization and reference checks can be done.

Table 8.2 shows the size of each of the components of the program restructuring system.

8.2. Examples and Experiments

8.2.1. Remarks about the Experiments

Before we discuss the experimental results, we would like to address several controversial issues. First, *what are "hard" problems and what are "simple" problems for parallel compilers?* Programs that a compiler faces can be roughly divided into two groups: programs with few data dependencies and programs with

† UNIX is a trademark of Bell Laboratories.

Table 8.2. *Sizes of components of the intelligent program restructuring system.*

Component	Language	Lines
Front-End & back-end	*C*	*18019*
ExpShell	*Prolog*	*3198*
Library for ExpShell	*Prolog*	*2543*
Hier-Blackboard	*Prolog*	*1612*
Machine knowledge manipulation	*Prolog*	*2097*
Program transformation system	*Prolog*	*4377*
Performance prediction	*Prolog*	*2162*
Intelligent program restructuring control	*Prolog*	*2353*
User interface	*C/Prolog*	*2058*
Total		*38419*

many dependence relations. The former is normally considered to be simple because there is more parallelism presented in the program. The matrix-vector multiplication example shown in section 3.1 demonstrates that this task is not necessarily as easy as many people would like to believe. The decision trees for such programs tend to be large since there are many possible alternatives at each stage. The task of the compiler is to pick the most plausible solution path efficiently. This task is important because the performance of a program on an architecture depends not only on the degree of parallelism inherent in the program but also on the match between the program and the architecture.

On the other hand, programs with many dependence relations are often considered to be "hard," since good speedup is usually difficult to obtain for these programs. We noted that if the goal of the compiler is to find the best path among the applicable program transformations for the given program, the complexity of the decision making is actually simpler for programs with more data dependencies than those with few. We are not claiming that this kind of problem is easy. Really difficult problems to the compilers are the programs for which some critical information cannot be easily guessed by the compilers.

For the above reasons, we pick two problems to demonstrate our ideas. The first example is the matrix-vector multiply example that we described in chapter 3. It is used here to show the complex decision-making process and the power of the system. The second example is an LU-factorization problem which has constructs that are inherently sequential, and as such it is difficult to parallelize for distributed machines. We will show how our system parallelized it to obtain some satisfactory results.

Another issue is that the results we show below only demonstrate the quality of the implementation of the framework. An important question is "what are the advantages of our framework over existing models?" Clearly, the same heuristics may be encoded in other systems to produce similar results. What cannot be shown in the experimental data is even more important to us. First, our framework allows us to implement multiple target parallel compilers; the system knowledge can be reused and transferred between different target machines. For example, all heuristics that we used in the second example below except those that are related to message passing and array distribution are collected from our experience on the shared-memory machines. They are transferred over and reused without reprogramming because the heuristics are encoded based on the machine features. Secondly, the hier-blackboard approach allows us to decompose the optimization problem into smaller but specialized modules whose interactions are specified in a higher level blackboard subsystem. Programming these modules is easier because the modules are more focused and less complex than the original problem. Programming the interaction among the modules is simpler because we are working on a higher level abstraction. Third, our system provides a heuristic analysis tool that can help to distill machine features from heuristics at the knowledge-acquisition phase. Regrettably, some advantages of this framework are not demonstrated because of time constraints. For example, our framework is designed to be integrated with self-learning modules, but we have not finished a learning module to show the result. Our preliminary study about learning modules will be discussed in the next chapter.

8.2.2. The Matrix-Vector Multiply Example Revisited

The matrix-vector multiply problem is considered by most to be simple to parallelize, because no dependence relations are present. On the other hand, lacking dependence relation implies that there are many different ways to parallelize the program and presents a challenge to automatic parallelizing compilers to find the most effective path. In this example, we illustrate how a heuristic hierarchy may be applied to the program parallelism optimization process. Based on the subproblem decomposition we described in section 4.3, the program restructuring process starts by examining the rules on the top layer of the hierarchy. After the focus of the program is chosen, the transformation subgoals on the next layer are selected. The rules associated with the subgoal are utilized to choose the applicable transformations. Similarly, when a transformation is chosen, the rules associated with it are applied to decide the merits and methods of performing the transformation on the program focus.

The flow of control is decided by the rules in the heuristic hierarchy. We will illustrate the decision-making process of the system with the matrix-vector multiply example that we used to illustrate the complexity of the optimization process for different architectures in chapter 3. We will now examine how the methodologies discussed above can be used to guide the compiler to generate the programs that were shown in figure 3.1.

To simplify the discussion we assume that the result vector y has been previously initialized to zero. We seek to transform this program to programs suitable for three different machines: the BBN Butterfly, the Purdue Pringle, and the Alliant FX/8. The rules used in this example are listed in Appendix A.

8.2.2.1. Mapping onto the BBN Butterfly

The system starts the program optimization process by consulting the machine knowledge manipulation system for the list of machine features for the target machine. For example, the fact "parallelize outermost loop without blocking" is added into the knowledge base by rule a.1 (listed in Appendix A) because the Butterfly provides a mechanism, *GenOnIndex*, which can schedule the loops automatically. The system discovers, among other facts, that memory optimization dominates instruction minimization (rule a.5), locality is important, and local memory should be used whenever possible (rule a.6). These facts are added to the system's state space in the working memory.

Next, the transformation heuristic hierarchy is used to optimize the program. First, the parallelism-matching control layer directs the restructuring of the program. In this example, it is trivial to select the program focus. By rule b.1, the whole subroutine is chosen as the program focus, since the original program consists only of a single statement inside the doubly nested loop.

The next step is for the program-restructuring control layer to decide which sequence of program-restructuring subgoals to achieve. Due to the simplicity of the dependence graph of this program, none of the transformations which are used to break the data dependence cycles are needed. Thus, the parallelism improvement subgoal is skipped (rule c.1). For the sake of flexibility, it is best to do processor assignment toward the end of the transformation process. However, array decomposition can be done only after tasks are created. So there is a conflict in deciding which of the two subgoals, *task creation and processor allocation subgoal* or *memory access optimization subgoal*, should be done first. Our solution to this problem is as follows. First, we find the tentative process allocation scheme and block the outermost loop to create "processes." The newly created outermost loop is marked, but is not actually parallelized. The loop instances of this marked loop form the tentative processes, and this information will be used to guide the array decompositions in the memory access optimization subgoal. The actual processor allocation is carried out at the end of the transformation process if the marked loop remains marked till then. This heuristic is encapsulated in the default ordering of rules c.4, c.5, and c.7.

After the task creation and processor allocation subgoal is picked, the system concentrates its restructuring efforts on the loop structures. At this stage, applicable transformations include loop interchanging and loop blocking (to create processes). According to the heuristic (rule e.1), if the program focus is a nested loop, then loop interchanging is checked to find the best order of the loops before the processes are created.

Therefore, the control goes down to the lower level transformation layer, and rules associated with loop interchanging are applied. We assume that the arrays in the Butterfly are stored in row order. There are no dependence relations that prevent us from interchanging the loop, so the loop is interchangeable. However, if loop j is changed to be the outermost loop, the array a will be accessed in columns no matter how we block the outer loop to form processes. This is not attractive because it increases the inter-task communications significantly. Therefore, based on the rules associated with loop interchange, the system decides that the

original loop order is the best and that no loop interchange is needed.

The next step is to find a tentative way of allocating the processes to the processors. Since the Butterfly has an instruction, *GenOnIndex*, that can schedule the loops automatically, we can parallelize the outermost loop without blocking (rule a.1). As a result, the outer loop i is marked to form tasks (rule e.4). There are n instances of the loop i, so n tasks are formed if each loop instance is viewed as a task. This information will be used to guide the array decompositions when the memory access optimization subgoal is pursued.

After the processor allocation phase, rule c.3 chooses the memory access optimization subgoal. Since local memory access is faster than global memory access on the Butterfly, locality is important (rule a.6). Also, the Butterfly supports a "block-transfer" instruction, which allows a block of memory to be transferred to, or from, the local memory to speed up the data transfer. This makes copying array references inside of loops into local memory beneficial. In the matrix-vector multiply program, there are two array references in the nested loops. Each element of array x is accessed once by every instance of the loop j. Also, elements of the i-th row of the array a are accessed exclusively by loop instance i. Since loop i is marked to be parallelized in the processor allocation subgoal, every processor that runs loop instance i will have to access every element of the array x and the i-th row of array a once. Rule f.1 suggests we copy array x and array a into local memory with block transfer operations. Since the i-th iteration accesses only the i-th row of the array a, there is no need to copy the whole array. The block transfer operation on array a is later changed by rule f.2 into a block transfer operation on row i of the array a in loop i. This gives us (by applying rule f.3):

```
for i in [1 .. N] do
    block_transfer(x, x_local, sizeof(x));
    block_transfer(a[i, *], a_local, sizeof(a[i, *]));

    for j in [1 .. M] do
        y[i] := a_local[j] * x_local[j];
    end for
end for
```

Since the block transfer statement of copying array x does not depend on loop i, it can be moved outside loop i to form another parallelized loop of P instances, where P is the number of the processors (rule f.4). In this way, the array is copied P times instead of N times, as it was in the original form.

After the memory optimizations are complete, the parallelism-improving subgoal is tried to see if there is any chance to improve the program further. It is relatively easy for the system to recognize that the inner loop j is an inner-product operation (rule d.1), so the loop is replaced by an inner-product operation (rule d.2). The final step involves the processor allocation subgoal again. Since no transformation that might prevent the parallelizing of the outermost loop i (which is marked for parallelizing) has been performed, the loop is directly parallelized as shown below.

```
forall k in [1 .. P] do
    block_transfer(x, x_local, sizeof(x));
end forall

forall i in [1 .. N] do
    block_transfer(a[i, *], a_local, sizeof(a[i, *]));
    y[i] := inner-product(a_local[*], x_local[*]);
end forall
```

8.2.2.2. Mapping onto the Pringle/CHiP Architecture

The Pringle/CHiP architecture consists of an array of 64 processors which communicate with each other via a packet-switched message network. There is no shared memory, and each processor runs one process. The communication pattern of messages between processors, defined at compile time as a communication graph, is used to configure the switch network at load time. Each of the memory modules is dual ported. One port goes to the processor while the other goes to a global bus; this allows the local memory of each processor to be a page of the global address of the front-end host. Programs and data are down-loaded to each processor and the results of a computation are loaded to the host over this bus.

For the same reason as in the case of the Butterfly, the system decides not to change the original order of loops after the rules in the transformation module, *loop interchange*, are used to decide the order of loop headers. The program-restructuring task is different here because the process creation time on the Pringle is expensive, and no self-scheduling primitive is available. The best strategy for processor allocation on the Pringle is to create P processes to run on the P processors that the Pringle has (rule a.2). So the n instances of the outermost loop i are blocked to form P tasks (rule e.3). The result is shown below:

```
forall k in [0 .. P-1] do
    for i in [k*n/P .. (k+1)*n/P] do
        for j in [1 .. m] do
            y[i] := y[i] + a[i, j] * x[j];
        end for;
    end for;
end forall;
```

Next, the memory access optimization subgoal is invoked to allocate the data. Since the Pringle is a non-shared memory machine, all the data must be distributed among the processors. Array decompositions are done by means of inter-process dependence analysis. By checking the bounds of the loops, the system discovers that the processor which runs process k (k-th iteration of the forall loop) accesses only rows $k*n/P$ to $(k+1)*n/P$ of the array a. In terms of the dependence relations, this means that no out-of-bounds dependence (dependence edge that has only one end in the loops) or cross-iteration dependence (dependence whose source and sink are in different loop iterations) of the array a exist. It is best to store these rows of the array in the local memory of the processor that runs the task. By rule f.11, the array a is divided into P blocks according to the memory access pattern, and the P blocks are allocated to local memories in the corresponding processors. Similarly, array y can be blocked into P "chunks" and stored in the local memories of the processors. Therefore, each of the processors computes n/P components of the y vector.

Since each process uses all the elements of array x, each processor needs to access the whole array x no matter where the array is allocated. If we are free to allocate the array x anywhere, the most direct method is to put it in one processor, say PE0, and then "broadcast" it to other processors by means of a pipeline process (rule f.12). To accomplish this, each element of x is passed from one processor to the next by using a "channel" variable. This transformation is termed "pipelining," which is a modified version of the transformation "scalar expansion" to pass the data through "channel_variables" instead of temporary variables. The channel variable $Ch\_x[k]$ implements a communication channel between processor k and processor $k+1$. Processor $k = 0$ reads the value of $x[j]$ and puts it in $Ch\_x[0]$. Processor $k=1$ reads the value in $Ch\_x[0]$ and puts it into $Ch\_x[1]$, etc. This approach is possible because the ratio of computation and communication is about 1 for Pringle. The overhead of this approach is the initial setup time for the pipeline, p pairs of write-read operations. The result of the transformation is shown below:

```
forall k in [0 .. p-1] do
    local tmp;
    for j in [1 .. m] do
        tmp = if (k==0) then x[j] else Ch_x[k-1];
        Ch_x[k] = tmp;
        for i in [k*n/p .. (k+1)*n/p] do
            y[i] = y[i] + a[i,j] * tmp;
        end for;
    end for;
end forall;
```

On some non-shared memory machines it is too costly to send a message consisting of only one word (for example, the Intel iPSC/2 and the nCUBE 2). In this case, it is best to broadcast large segments of the x vector by using a tree structure.

8.2.2.3. Mapping onto the Alliant FX/8

In the case of the Alliant FX/8 there are three important programming issues. First, because of the powerful vector instruction set in each processor, one should exploit as many vector operations as possible. Second, since cache access is twice as fast as a memory access, the programmer must force as many memory

accesses to be from the shared data cache as possible. Third, because only one operand in a vector instruction may come from memory or cache, it is important to keep vector operands that are used repeatedly in vector registers.

Most parallel compilers can recognize the inner-product operation in the original matrix vector multiply program and translate the program into the following form:

```
for i in 1 .. n do
    y[i] = inner_product(A[i, *], x);
```

Although the Alliant supports fast inner-product operations, this transformation does not really utilize the parallelism capabilities of the Alliant FX/8. The array x is accessed n times on each processor; thus the array x needs to be brought into the cache repeatedly. Since each vector register in the Alliant FX/8 can hold only thirty-two words of data, the vector x and the matrix a in the sample program need to be loaded into the vector registers repeatedly. This data traffic floods the bus and slows down the computations significantly.

In general, without intelligent program analysis, this communication bottleneck problem is hard to solve. Our system tries to improve the matching between the program and the computational model of the Alliant by examining and managing the memory accesses intelligently.

As in the case of the Butterfly, task creation and processor allocation is the first subgoal selected. Since the Alliant has a vector capability, both the vector processing parallelism in the innermost loop and the multiprocessing parallelism in the outermost loop need to be explored. Before the outer loop is blocked to form tasks and the inner loop is blocked to form vector operations, loop interchange is considered to find the best ordering of the loop headers (rule e.1). Thus control goes down to the transformation layer, and the rules associated with the transformation "loop interchange" are applied. First, the nested loops i and j in the original source are checked, and as before, it is concluded that they are interchangeable. Next, rules relating to loop orders are applied to decide the best order of the loop headers. Program size matching and memory utilization matching indices can be used to select the loop order. Rule a.5 suggests that memory optimization dominates the instruction minimization, so memory optimization matching is considered.

The matrix-vector multiply program accesses vector x in total n times, once for each loop instance of loop i. Loop j is the loop that scans through vector x. If loop j is the inner loop, and loop i is the outer loop, then each value of the vector x will be accessed once by every loop instance of loop i. Therefore, the vector needs to be brought into the cache repeatedly. On the other hand, if loop i is the inner loop and loop j is the outer loop, the value $x[j]$ is brought into the cache and used by all loop instances of the inner loop i for each loop instance of the outer loop j. In this loop order, the network traffic for references of vector x is decreased significantly. Therefore, the loop order where loop j is outside is preferred according to the memory allocation matching function. In other words, the loops need to be interchanged.

After the loops are interchanged, the innermost loop is blocked to form vector operations, and the outermost loop is translated into tasks and may be blocked to form processes. For the vector loop blocking, the inner loop i is blocked according to the vector register size of the Alliant (rule e.2). The vector operation is created by vectorizing the innermost loop after the blocking. The resulting program is shown below. Each loop instance of the outermost loop j forms a task. Since the Alliant instruction set can automatically allocate the processes to the eight processors, no loop blocking is needed to match the number of processes with the number of processors (rule a.1). Subsequently, loop j is marked to be parallelized.

```
for j in [1 .. m] do
    for k in [0 .. n/32-1] do
        k1 = k * 32 + 1;
        k2 = (k+1) * 32;
        y[k1 .. k2] sum= a[k1 .. k2, j] * x[j];
    end for;
end for;
```

The next step is to optimize memory access. Rule a.7 suggests that keeping one vector operand in a vector register is beneficial. Since vector segment $y[k*32+1 .. (k+1)*32]$ is used repeatedly by each instance of the outer loop j, it is best to keep this segment in the vector register. This can be accomplished by interchanging loops j and k (rule f.13). Note that in the previous task creation and processor allocation subgoal, the loop j is marked as "to be parallelized." However, according to rule f.14, the utilization of vector registers and

vector operations is weighted to be more important. So the previous decision is revoked, and the loops are interchanged. Loop *k* becomes the outermost loop and is thus parallelized. The resulting program is:

```
forall k in [0 .. n/32-1] do
    local k1, k2 : int;
    k1 = k * 31 + 1;
    k2 = (k+1) * 32;
    for j in [1 .. m] do
        y[k1 .. k2] sum= a[k1 .. k2, j] * x[j];
    end for;
end forall;
```

In the final version, each 32 word y vector segment can be saved in a register for the lifetime of the process and can be written to memory only at the end of the computation. Experiments performed in collaboration with Dan Sorensen [Sore] at the Illinois Center for Supercomputer Research and Development have shown that this implementation of the program is the fastest version of a matrix-vector multiply available for the machine.

The matrix-vector multiply example described above served three purposes:

1. It demonstrated how the inference engine works.

2. It illustrated that a different sequence of transformations was required to produce an optimal program for each machine.

3. It showed the complexity of the program parallelism optimization process.

Many heuristics were needed even for this simple program. This reinforces our view that an expert systems approach is a more flexible and extensible approach than the conventional hard-wired heuristics approach.

On the other hand, the example described above is far too simple to illustrate many of the most interesting and important issues in program restructuring. In particular, it fails to illustrate the issues relating to the introduction of synchronization needed in many problems to satisfy data dependence constraints between parallel tasks. This topic is considered in the next example.

8.2.3. A More Realistic Example: LU-Factorization

Our next experiment concerns parallelizing programs for distributed-memory parallel computers. The example that we choose involves solving a system of linear equations with Gaussian elimination. This example was used in [Karp87] as a realistic example to show "the state of the art of parallel programming and what a sorry state that art is in"†. Here we chose the nCUBE 2, a distributed-memory architecture, to show how heuristics and architecture properties influence the decision making of the compiler.

The procedure for solving the linear system is:

1. *LU factorization:* Use Gaussian elimination to compute two triangular matrices L and U such that $A = LU$. The original problem becomes: $LU\ x = b$ or $L\ y = b$ where y is a solution to $U\ x = y$.
2. *Forward elimination:* Solve the lower triangular linear system $L\ y = b$ by forward elimination.
3. *Back substitution:* Solve the upper triangular linear system $U\ x = y$ by back substitution.

Since the time complexity for the three steps is $O(N^3)$, $O(N^2)$, and $O(N^2)$, respectively, the LU factorization consists of the major time in solving the linear system. Therefore, we will only discuss step 1 factoring the matrix A into its LU components. For linear systems that have small or zero diagonal elements, Gaussian elimination may generate disastrous results. To avoid computational errors, a technique called *partial pivoting* is used to interchange the rows of the matrix so that the largest remaining element in the kth column is used as the pivot.

In the program shown in figure 8.5, the following procedure is applied to each column k in turn:

a) *Find pivot:* Compare the elements on or below the diagonal of the current column and find the row index of the element whose absolute value is the maximum. This element is called the pivot.

† Karp derives different versions of this program for several models of parallel architecture and explains difficulties encountered for each models.

```
procedure lu_factorization(n, tolerance, a) returns: (a, ipvt, info);
param
      n: integer;                -- sizes of the rows and columns
      tolerance: real;           -- minimize size for acceptable pivot
      a : array[1..n, 1..n] of real;     -- array to be factorized
      ipvt: array[1..n] of integer;      -- record pivoting rows
      info: integer;                     -- how many nonzero pivots do we have
var
      rmax, t : real;
      imax, in: integer;
begin
   in = 0;   info = 0;
   for k in 1 .. n-1 loop
      imax := k; rmax := abs(a[k,k]);              --- find pivot row
      for i in k+1 .. n loop
         if (abs(a[i,k]) > rmax) then
            imax := i; rmax := abs(a[i,k]);
         end;
      end;
      if (rmax < tolerance) then in = k; end;
      ipvt(k) := imax;

      if (in != k) then
         info := info + 1;
         if (imax != k) then
            for j in 1 .. n loop        --- interchange row k and imax
               a[k,j], a[imax,j] := swap(a[imax,j], a[k,j]);
            end;
         end;

         t := -1.0 / a[k,k];
         for i in k+1 .. n loop         --- scale the k-th column
            a[i,k] := a[i,k] * t;
         end;

         for i in k+1 .. n loop         --- apply multipliers to rows
            for j in k+1 .. n loop
               a[i,j] := a[i,j] + a[i,k] * a[k,j];
            end;
         end;
      end;
   end;
end;
```

Figure 8.5. *The LU factorization program represented in Blaze.*

b) *Move pivot to diagonal:* Interchange the diagonal row and the row that contains the pivot so that the pivot element will be moved to the diagonal.

c) *Compute the multipliers:* Divide the elements below the pivot by the pivot to produce a set of multipliers.

d) *Apply multipliers to all rows below the diagonal.* For each row below the diagonal, multiply the elements to the right of the pivot with the multipliers of that row and subtract the product from the corresponding part of the row.

Some obstacles in parallelizing this procedure are as follows:

1. The procedure is inherently sequential in the sense that the kth iteration of the outermost loop cannot be performed until after the $k-1$th iteration is finished.
2. On a distributed-memory machine, the cost of the procedure is very sensitive to the distribution of the array a.
3. Steps (a) and (c) operate on columns and steps (b) and (d) operate on rows. None of the steps (a) - (d) can be overlapped. The mix of sequential and parallel parts presented in this example is typical of many numerical procedures. We deliberately chose this "non-perfect" algorithm to see how an automatic parallel compiler can utilize some "generic" heuristics (as opposed to heuristics restricted to a particular example) to parallelize the program.

The SPMD (single program, multiple data) model is used for distributed-memory machines because it is convenient to have only one program for all the processors. (The control flow of the program in each processor is normally decided by the processor identifier.) To generate programs for the distributed-memory SPMD model, we assume that the outermost loop will be distributed across the processors either in blocks or in cycles. Initially, a pair of I/O statements will be generated for all cross task dependence relations. The I/O statements between two processors may be merged with the message consolidation algorithm we described in chapter 6. Computation that uses only local data of a particular processor will be done on that processor only. For computation that needs data from more than one processor, the distribution of the computation depends on the cost of getting the external data and the cost of distributing the results to the processors that use them. The synchronization points of the program are taken into consideration. The computation will be duplicated in all processors that use the data if the computation can be done locally before the expected arrival time of the data when the data are computed by other processors.

The process of the transformation that the system went through will be explained below:

In the LU factorization program, the outermost loop k is distributed into processors in cyclic distribution because the cross-iteration dependence relations are of distance one. (This means that if the loop is distributed in blocks the execution will have to be serialized.)

One of the most important decisions for the compiler to make is how to distribute the data; for this program, the array a is the most critical array in the subroutine. There are many ways to distribute a non-sparse array on distributed-memory computers; most commonly used methods include *block distribution* (each processor gets a chunk of the array, either row or column), *cyclic distribution* (the rows or columns of the array are distributed to the processors in a round-robin scheme, for example, processor i gets column i, $p+i$, $2p+i$, ...), or *hybrid distribution* (applying either block or cyclic distribution to each dimension of the array; such as block-block or cyclic-block distribution on a 2-dimensional array).

The objective is to distribute arrays so as to minimize the communication between processors. A heuristic for distributing arrays is that if the index of the parallel loop appears as simple subscripts (contain no operations) in all references of an array, then the array is to be distributed based on the distribution pattern of the loop. As stated above, the LU factorization program is decomposed into four steps with data dependencies between each of these four steps. For step (d), it does not matter which subscript we choose to distribute, because the shape of the array a used in step (d) is symmetric. For steps (a) and (c), the computation will need data of size $\frac{n-k}{p}$ and $\frac{n-1}{p}$ from every processor respectively; but it allows the computation to be executed in parallel if array a is distributed in rows. If the array a is distributed in columns, then all the data used in the computation belong to the processor ik that has the column k. This means that the computation can be done sequentially in processor ik without reading from other processors. For step (a), *imax* and *in* have to be sent to all other processors. For step (d), elements of array $a[k..n,k]$ need to be distributed to all processors in both kinds of array distributions. For step (c), distributing by rows would involve two processors exchanging data of size n with each other, whereas the column distribution will allow all processors to work on their own data in parallel. As a result, cyclic distribution of the columns is chosen.

Since the array a is distributed in columns, the process of selecting the pivot in step (a) involves only local data of processor $ik = k \bmod p$ at the kth iteration. So the computation is localized to processor ik and the values *imax* and *in* are broadcast to all processors by processor ik. Similarly, scaling the column k at the kth iteration of the outermost loop in step (c) involves only local variables in processor ik but the result is needed by all processors. So the computation is again restricted to processor ik and the column $a[k..n,k]$ is then broadcast to all processors by processor ik. The resulting parallelized program is called $P1$ and is shown in figure 8.6.

For the program shown in figure 8.6, there are two broadcasting statements issued by processor *ik*, one intuitive way to improve the performance is to merge the two broadcast statements. This move is non-trivial for the compiler since the dependencies in step (b) (which swaps rows *k* and *imax*) prevents the two broadcasting statements from been merged. Although this restriction can be overwritten through user interaction, the speedup results from merging the two broadcasting statements is not very significant. This is due to the overhead of copying the data into a temporary buffer and the additional data synchronization involved. This version of the program is called $P2$ and its result is shown in in column $P2$ of tables 8.4 and 8.5.

The current nCUBE 2 compiler produces very slow code for the calculation of addresses of multi-dimensional array references. This implies that whenever there is a choice, making array references of higher dimensions loop invariant is better than making array references of lower dimensions loop invariant. When applied to our example, in step (d) w[i] is loop invariant in loop j, and hence can be replaced by a scalar variable *tmp* through vector scalarization. On the other hand, if we interchange loop *i* and loop *j*, the reference $a[k,j]$ becomes loop invariant to the new inner loop *i* and can be scalarized. The above heuristic says that interchanging the loops *i* and *j* and scalarizing the reference $a[k,j]$ is better than scalarizing reference $w[i]$ in loop *j*. Additional speedup of about five percent was obtained through this transformation. The resulting program is called $P3$ and is shown in figure 8.7.

The sequence of the transformations that are applied in the above example is listed in table 8.3.

We translated the final optimized Blaze program into Fortran and tested it on a 64 node nCUBE 2. The nCUBE 2 we used has 4 mega-bytes of memory in its first 16 processors and 1 mega-byte in the rest of processors. We ran the tests with different sizes of array *a*, starting with 100 and doubled the size at each step to the largest size which would fit on the machine (800 for cube of dimension ≤ 1, and 1600 and 3200 for larger cubes). For sequential cases, we can only run the program with an array of size 800, so only the speedups for array of size up to 800 are reported. The timing is reported for all tests.

We can see that the speedup is slowly going down for cubes of larger size. The cost of broadcasting increases as the size of the cube increases and the operations between communication points decrease. Considering that the program at hand has sequential control flow, the speedup we obtain is very satisfactory. The rules that are used in this example are listed in Appendix A.

8.3. Summary

In this chapter, we presented a prototype intelligent parallel programming environment and its components. Two examples and experimental results were also given.

```
procedure lu_factorization(n, m, tolerance, a, p, pid) returns: (a, ipvt, info);
param
   n, m: integer;                  -- sizes of row and columns (m=n/p)
   tolerance: real;                -- minimize size for acceptable pivot
   a : array[1..n, 1..m] of real; -- array to be factorized
   ipvt: array[1..n] of integer; -- record pivoting rows
   info: integer;                  -- how many non-zero pivots do we have
   p, pid: integer;                -- number of processors and processor id
var
   w: array [1..n] of real;              -- temporary array to hold column k
   rmax, t: real;
   imax : integer;
begin
   info = 0;     maxc = (n+p-1) / p;
   if (pid > mod(n-1, p)) then maxc = maxc - 1; end;
   for k in 1..n-1 loop
      --    ik = processor that has the column k of a0.
      --    ij = column number of k (of a0) in a of processor ik
      ik := mod(k-1, p);  ij := (k+p-1) / p;
      if (pid == ik) then
         imax := k; rmax := abs(a[k,ij]);   --- find pivoting row
         for i in k+1..n loop
            if (abs(a[i,ij]) > rmax) then
               imax := i; rmax := abs(a[i,ij]);
            end;
         end;
         if (rmax < tolerance) then in := k; end;
         w[1] := imax;    w[2] := in;
      end;
      imax, in := broadcast(ik, w[1..2], 2);
      ipvt[k] := imax;

      if (in != k) then
         info := info + 1;
         if (imax != k) then                  --- interchange row k and imax
            for j in 1..maxc loop
               a[k,j], a[imax,j] := swap(a[imax,j], a[k,j]);
            end;
         end;
         if (pid == ik) then                  --- scale the k-th column
            t := -1.0 / a[k,ij];
            for i in k+1..n loop
               a[i,ij] := a[i,ij] * t;
            end;
         end;
         w[k..n] := broadcast(ik, a[k..n,ij], n-k+1);
         if (pid != ik) then ij := ij + 1; endif;
         for i in k+1..n loop           --- apply multipliers to rows
            tmp := w[i];
            for j in ij..maxc loop
               a[i,j] := a[i,j] + tmp * a[k,j];
            end;
      .....
end;
```

Figure 8.6. *The parallelized LU factorization program for distributed-memory computers.*

```
procedure lu_factorization(n, m, tolerance, a, p, pid) returns: (a, ipvt, info);
      ..............
--    same as in figure 8.7 except the last loop:
        if (pid != ik) then ij := ij + 1; endif;
       for j in ij..maxc loop          --- apply multipliers to rows
          tmp = a[k,j];
          for i in k+1..n loop
              a[i,j] := a[i,j] + w[i] * tmp;
           end;
      ..............
end;
```

Figure 8.7. *The optimized LU factorization program with last loop interchanged.*

Table 8.3. *Sequence of transformations applied in the example.*

☞	*Loop blocking(k, cyclic)*
☞	*Cyclic distribution of columns of array a*
☞	*Localize computations of finding pivot and scale kth column*
☞	*Convert cross-task dependencies into messages*
☞	*Change read-write pairs into broadcasting routines*
☞	*Consolidate messages for imax and in and messages for a[k..n,k'] (this gives us P1).*
☞	*Apply statement reordering and message consolidation to merge the above two broadcast calls (this gives us P2).*
☞	*Interchange loop i and j and then scalarize vector a[k,j] (this gives us P3).*

Table 8.4. *Test results for the LU_factorization (time in seconds) where S1 is the sequential program, and P1, P2, and P3 are the parallelized programs as discussed above.*

# of processors T^p	array size	S1	P1	P2	P3
1	50	0.335092			
1	100	2.596930			
1	200	20.332565			
1	400	160.742973			
1	800	1278.104350			
2	50		0.205675	0.199552	0.192359
2	100		1.418245	1.405851	1.351634
2	200		10.600395	10.575588	10.156595
2	400		82.200417	82.150757	78.856606
2	800		648.020020	647.920776	621.806885
4	50		0.141532	0.125691	0.119751
4	100		0.818316	0.786300	0.749784
4	200		5.648145	5.583807	5.336084
4	400		42.314747	42.184917	40.385593
4	800		328.515930	328.253601	314.585968
8	50		0.131961	0.096773	0.091504
8	100		0.564934	0.493654	0.466063
8	200		3.272318	3.129107	2.967397
8	400		22.621096	22.332933	21.280478
8	800		169.469940	168.888718	161.444214
16	50		0.130673	0.095632	0.090602
16	100		0.454801	0.383533	0.360316
16	200		2.145916	2.002870	1.883705
16	400		12.973400	12.687386	12.006158
16	800		90.642464	90.074875	85.726631
32	50		0.136305	0.094692	0.089864
32	100		0.409748	0.324679	0.303406
32	200		1.600865	1.429922	1.331797
32	400		8.199018	7.855660	7.359412
32	800		51.325867	50.639671	47.839237
64	50		0.144678	0.096747	0.091986
64	100		0.404622	0.306533	0.286258
64	200		1.364162	1.165164	1.077079
64	400		5.909894	5.512052	5.106190
64	800		32.021828	31.221504	29.192616

Table 8.5. *Speedup for the LU_factorization program.*

# of processors $\frac{T^1}{T^p}$	array size	P1	P2	P3
2	50	1.629231	1.679222	1.742014
2	100	1.831087	1.847230	1.921326
2	200	1.918095	1.922594	2.001908
2	400	1.955501	1.956683	2.038421
2	800	1.972322	1.972624	2.055468
4	50	2.367606	2.665998	2.798240
4	100	3.173505	3.302722	3.463571
4	200	3.599866	3.641345	3.810391
4	400	3.798746	3.810437	3.980206
4	800	3.890540	3.893649	4.062814
8	50	2.539326	3.462660	3.662048
8	100	4.596873	5.260628	5.572058
8	200	6.213506	6.497881	6.851987
8	400	7.105888	7.197575	7.553541
8	800	7.541776	7.567731	7.916694
16	50	2.564355	3.503974	3.698506
16	100	5.710036	6.771073	7.207368
16	200	9.475005	10.151715	10.797922
16	400	12.390196	12.669511	13.388377
16	800	14.100503	14.189355	14.909070
32	50	2.458398	3.538757	3.728879
32	100	6.337871	7.998454	8.559257
32	200	12.700987	14.219353	15.267015
32	400	19.605150	20.462057	21.841821
32	800	24.901759	25.239192	26.716654
64	50	2.316123	3.463591	3.642859
64	100	6.418163	8.471943	9.071991
64	200	14.904803	17.450389	18.877505
64	400	27.198959	29.162091	31.480019
64	800	39.913536	40.936669	43.781770

CHAPTER 9
CONCLUSION

9.1. Summary of the Thesis

In this thesis we have discussed issues related to the construction of intelligent parallel compilers for different parallel architectures.

First, we introduced a new program optimization model called the *feature-directed program optimization* model. Under this model, the program optimization process is driven by architectural features and program dependence graphs. The major differences between this model and existing models are the following:

1. Both features of the program and the machine influence the decision-making process. The model can respond well to different programs and architectures.
2. The whole decision tree of the program optimization process is considered. Thus, it is possible for the optimal version of a program for a target machine to be discovered.
3. Systematic state-space search algorithms (such as A^*) can be easily integrated for fully automatic program optimization and pruning non-promising branches in the decision tree.
4. Machine features are explicitly encoded in the heuristics. This allows systematic analysis of the heuristics and provides hooks for organizing knowledge and integrating self-learning modules.

Next, a framework for realizing this model into intelligent parallel compilers is introduced. The framework is based on the following essential components:

- *Flexible machine feature manipulation.* An object-oriented machine feature representation and manipulation scheme is designed. The machine feature manipulation model is the foundation for knowledge organization and generalization.
- *Accurate performance prediction.* A performance prediction model based on machine features and performance-characterizing factors has been designed. The prediction model is used in heuristic-based state-space search algorithms to compute heuristic functions, and in the rule-based system implementation to discard non-promising paths. The prediction model is highly flexible and can be adjusted to suit different objectives at different stages of compiling and accommodate different classes of parallel computers. Many useful evaluation functions for the performance characterization factors are listed in chapter 6.
- *Inference capability.* Inference capability and some AI techniques are integrated to improve efficiency of the heuristic-based search algorithms.
- *Effective run-time tests.* Run-time tests can help to optimize programs that fail static analysis. However, excessive run-time tests may do more harm than good to the performance of the program. A technique using constraint propagating and constant-folding to minimize run-time tests is introduced.
- *Modular knowledge encapsulation.* Under this framework, program transformations are viewed as intelligent modules that contain both heuristics and techniques about the transformations. Each module can evaluate its applicability and possible contributions based on the current program structures and hardware features. This makes it possible to dynamically decide the sequence of transformations based on features of the program and the target hardware.

A prototype intelligent program optimization system is built to demonstrate the ideas we have derived in this thesis. The system is implemented as a collection of expert systems and is based on the feature-directed program optimization framework and the hier-blackboard model that we discussed in chapter 4. The system integrates state-of-the-art AI technologies with advanced program-restructuring techniques. Twenty-one

program transformation techniques and a host of heuristics to utilize these transformations are implemented in the system. The system can dynamically select the most appropriate program transformation at compile time or generate minimal run-time tests to determine the control flow at run-time. New target machines and program transformation heuristics can be easily incorporated into the system. This allows the system to be used as a test bed for new program transformation heuristics or new hardware designs of parallel computers.

The prototype system is realized by the implementation of the following subsystems:

- A machine knowledge manipulation system as discussed in chapter 5.
- A hier-blackboard simulator that features hierarchical problem-solving control and opportunistic reasoning is constructed.
- An efficient performance prediction subsystem that can accurately predict the performance of programs and performance changes caused by program transformations.
- A knowledge base of transformation knowledge. Various heuristics for loop optimization, memory hierarchy utilization, program partitioning, scheduling, and interprocess communication and synchronization for shared- and distributed-memory multi-processor computers are included in the system knowledge base.
- A program transformation subsystem. Twenty-one program transformation techniques are included and the number is steadily growing.

Theoretical foundations of two program optimization techniques, *message consolidation* and *array reshaping*, are introduced. Message consolidation can be used to minimize communication cost for distributed-memory parallel computers. An algorithm that can find the optimal grouping of the messages and is dead-lock free is also presented. Array reshaping is a generic mapping to modify array storage patterns. It can be applied in many different ways to improve the data storage and communication processors. Some of these methodologies were exploited in chapter 6.

Different AI techniques to increase the degree of intelligence and improve the efficiency of compilers are also discussed.

9.2. Contributions

Our contribution to the field of parallel compiling can be itemized as follows:

1. We have laid down a foundation for systematic and automatic program optimization.
2. We have developed a practical framework for the construction of multiple-target parallel compilers.
3. We have formulated the program optimization problem into the planning problem and have derived several systematic algorithms for optimizing parallel programs.
4. We have developed a hierarchical blackboard problem-solving model which is highly parallel and flexible and is suitable for program optimization.
5. We have designed a new machine classification and knowledge manipulation scheme. This scheme allows better organization of the program-restructuring heuristics and makes porting the system to new parallel machines easier.
6. We have designed an accurate, efficient, and flexible performance prediction model to estimate the performance of programs on different parallel computers. The performance prediction system features symbolic processing and forms a basis for the decision making of intelligent parallel compilers.
7. We have studied two program transformation techniques, *messageconsolidation* and *arrayreshaping* and their application in improving program parallelism. We gave a theoretical foundation for these two transformations and derived conditions and algorithms for utilizing these two transformations profitably on distributed-memory architectures.
8. We have built a prototype parallel programming environment to demonstrate the ideas and to experiment with different heuristics.

9.3. Future Work

The following problems are logical follow-up of this research and should be probed further.

1. Study the orchestration effects of linking several program transformation techniques in the systematic program optimization process.
2. Explore the parallelism in the hierarchical problem-solving model.
3. Integrate self-learning modules to improve the intelligence level of the system.

9.3.1. Chaining Multiple Program Transformations

Although heuristics may combine multiple transformations to achieve certain specific goals, the procedure we used in chapter 3 does not address this problem explicitly. Instead, the group of the transformations is treated as a new transformation. The lack of systematic treatment of the accumulated effects of program transformation is a potential problem for the algorithm. One possible way of addressing this problem is to allow several steps of look-ahead. This will allow the system to discover accumulated effects not specified in the heuristics. On the other hand, the known relation between the transformations in the heuristics should not be abandoned so that a representation scheme to integrate the knowledge without lumping the multiple transformations into a new transformation should be developed.

9.3.2. Parallel Execution of the Compiling Process

To execute the compiler in parallel is not difficult under our model. To achieve this, we need to implement the hier-blackboard architecture instead of using the simulator.

9.3.3. Self-Learning Modules

The essential characteristic for a system to be intelligent lies in the ability to learn -- the ability to enhance its capability through the acquisition of new knowledge. Depending on the degree of human assistance, the learning of software systems can be classified as knowledge acquisition (with help from knowledge engineers) or machine learning (self-learning).

9.3.3.1. Knowledge Acquisition

After decades of intensive studies by knowledge engineers and psychologists, knowledge acquisition is still in a state that is more an art than a science. On the other hand, past research has given us some principles, methodologies, and advice for conducting knowledge acquisition.

Knowledge acquisition is the activity of gathering information from any source. Techniques in knowledge acquisition depend on the source of the knowledge. If the source is a human expert, then the (sub-)task is called knowledge elicitation. If the source of the knowledge is the documents or literature, then the process is called literature summarization. Knowledge elicitation is difficult because it involves not only the skill of the knowledge engineers and domain experts, but also human interaction and psychological problems. Literature summarization is a very time-consuming task, but it can also yield a great deal of knowledge and is effective when no experts are available. We build up the knowledge base mainly from literature summarization and our own experiments.

The problem domain of optimizing parallel programs for different classes of parallel computers has the following characteristics:

- *Bound to hardware architecture.* Most heuristics are developed from a particular type of architecture and need some analysis before they can be applied to other types of architectures or simply be encoded.
- *Partial and fragmented knowledge.* Most experts have only partial knowledge about the optimization. They are familiar with only a few types of architectures and only few types of problems such as solving PDE problems or computer graphics problems on parallel computers.
- *Not program transformation oriented.* Decomposing the process of program optimization into program transformations is suitable for a compiler or automatic program optimization but is not the natural way that experts attack the problem. Raw heuristics need to be examined and associated with program transformation techniques before they can be applied.

9.3.3.2. Knowledge Refinement

The development of an expert system is typically an iterative process. It consists of knowledge acquisition-refinement cycles that are repeated until the intended performance of the system is achieved. In the refinement cycles, the machine's performance is compared with the performance of the human expert, and the machine representation of the knowledge is compared with the knowledge of the original expert. This refinement cycle extends the role of expert systems to expert support systems since both man and machine learn through repeated knowledge acquisition-refinement cycles.

9.3.3.3. Self-Learning Modules

Based on the above understanding, the following self-learning modules are being studied.

- *Generalization of heuristics by relaxation.* This is a simple systematic way of generalizing heuristics. A heuristic may be generalized by systematically relaxing the conditions of the heuristics one by one. The effects of the resulting knowledge are then tested with a library of programs. A limit to this approach is that only minor relaxation may produce meaningful results, since the heuristics are supposed to have been studied extensively by the knowledge engineer.
- *Neural networks.* The design of the system makes a neural network implementation an ideal candidate for learning. The input to the neural network is a set of states of the system that includes the list of predicates that encode the machine features and the program features, the weights of the evaluation functions, the certainty factors in the rules, the program transformations under consideration, and a feedback of the results of applying the transformations. After the system gets to an equilibrium state, the weights of evaluation functions and certainty factors of rules derived by the neural network can be used to solve the particular types of programs and the target architecture represented in the input. The input data can be collected through the history files generated by the system.
- *Causal-based learning.* Causal-based learning is an incremental learning method that learns from examples and instructor guidance. The module learns by watching an instructor perform optimization on the programs. It uses an underlying model to explain some behavior of the experts and to use that explanation as a hint for learning. Expert guidance can be used to increase the learning efficiency, to permit continuous learning and reduce system brittleness. The objective of the learning is to learn relationships between features of the program and machine and the program transformation sequences.
- *Model-based learning.* Model-based learning is a generate and test learning model. The system learns by examining examples generated automatically by a qualitative model. In our case, the system can generate sequences of program transformations and validate the quality of the sequence by measuring the performance of the resulting programs.

9.4. Closing Remarks

Will users be freed from tedious parallel program optimization in the near future? Judging from the rate of progress of the field and the nature of the problem, the answer is probably a *no*. This prompts us to search for alternative methodologies for systematic program optimization to utilize existing technologies and to find better ways to analyze and integrate heuristics. There will be always programs that will require extensive user interaction because of the data-dependent nature of the programs. Our goal is to search for methodologies that may replace the human programmer with a software system in the guess-test cycle of the program optimization process and find a more efficient framework for such process.

The results that we have reported in this thesis are still somewhat primitive and warrant further study. We have demonstrated the importance and potential of systematic analysis and AI techniques in parallel compilers. However, the realization of real intelligent parallel compilers requires joint efforts from both compilers and AI researchers. We believe this work will evolve into a multiple discipline, coordinated research that has a high potential for re-shaping the focus of future research and parallel compiler systems. Our results in this thesis provide a foundation for further probes and give a very practical implementation model as a start.

BIBLIOGRAPHY

[AbKw85] W. Abu-Sufah, and A. Kwok, "Performance Predication Tools for Cedar: A Multiprocessor Supercomputer," in *Proceeding of the 12th International Symposium on Computer Architecture*, 1985, 406-413.

[AhSeUl86] A. Aho, R. Sethi, and J. Ullman, "Compilers: Principles, Techniques, and Tools," Addison Wesley, 1986.

[ABCCF88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN analysis system for Multiprocessing," in *Proceedings of the 1987 International Conference on Supercomputing*, LNCS, February, 1988.

[Allen74] F. Allen, "Interprocedural Data Flow Analysis," in *Information Processing 74*, North Holland Publishing, Amsterdam, 1974, 398-402.

[Allen86] F. Allen "Compiling for Parallelism," in G. Almasi, R. Hockney, and G. Paul, editors, *Proceedings of the IBM Institute Europe*, North-Holland Press, 1986.

[Allen83] J.R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph.D. Thesis, Rice University, Houston, Texas, April 1983.

[AlBaKe86] J.R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield, "PTOOL: A Semi-Automatic Parallel Programming Assistant," in *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986, 164-170.

[AlKe84a] J.R. Allen and K. Kennedy, "A Parallel Programming Environment," technical report, TR-84-3, Department of Computer Science, Rice University, July 1984.

[AlKe84b] J.R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," in *Supercomputers: Design and Applications*, IEEE Computer Society Press, Silver Spring, MD. , 1984, 186-205.

[AlKe87] J.R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," in *ACM Transactions on Programming Language and Systems*, Vol 9, No. 4, October 1987.

[ASKL79] W. Abu-Sufah, D. Kuck, and D. Lawrie, "Automatic Program Transformations for Virtual Memory Computers," in *Proceedings of the 1979 National Computer Conference*, June 1979, 969-974.

[Anna90] M. Annaratone et al., "The K2 Parallel Processor: Architecture and Hardware Implementation," in *Proceedings of the 17th Symposium on Computer Architecture*, Seattle, June 1990.

[Banc76] U. Banerjee, "Data Dependence in Ordinary Programs," Technical Report, Department of Computer Science, University of Illinois at Urbana-Champion, Rpt. No. 76-837.

[Banc88] U. Banerjee, "Dependence Analysis for Supercomputing," Kluwer Academic Publishers, Norwell, Mass., 1988.

[Barth78] J. Barth "A Practical Interprocedural Data Flow Analysis Algorithm," in *CACM* Vol 21. No. 9, September 1978, 724-736.

[BCJMMKSW90] K. Birman et al., "The ISIS System Manual, Version 2.1," The ISIS Project, Department of Computer Science, Cornell University, 1990.

[BDHLW87] M. Byler, J. Davies, C. Huson, B. Leasure, and M. Wolfe, "Multiple Version Loops," in *Proceedings of the International Conference on Parallel Processing*, 1987, 312-318.

[BeDeWe85] J. Beetem, M. Denneau, and D. Weingarten, "The GF11 Supercomputer," in *IEEE Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Mass. June 1985, 108-113.

[BeLaLe88] B.N. Bershad, E.D. Lazowska, and H.M. Levy, "PRESTO: A System for Object-oriented Parallel Programming," in *Software - Practice and Experience*, Vol. 18, No. 8, 1988, 713-732.

[Bern86] A.J. Bernstein, "Analysis of Programs for Parallel Processing," in *IEEE Transactions on Computers*, 746-757, October 1986.

[BWJALG90] F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, and D Gannon, "Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000," in *Proceedings of the 1990 International Conference on Supercomputing*, 1990, 401-413.

[Borkar88] S. Brokar et al., "iWarp: an Integrated Solution to High Speed Parallel Computation," in *Proceedings of Supercomputing 88*, November 1988.

[BrMcWe85] W. Brantley, K. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," in *Proceedings of the 1985 International Conference on Parallel Processing*, 1985, 782-789.

[BuCy86] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," in *ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN, Notice*, Vol 21, No 7, July 1986, 162-175.

[CCHKT88] C.D. Callahan, K.D. Cooper, R.T. Hood, K. Kennedy, and L. Torczon, "Parascope: A Parallel Programming Environment," in *The International Journal of Supercomputer Applications*, Vol 2. No. 4, Winter, 1988, 84-99.

[CaKe88] C.D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," in *The Journal of Supercomputing*, Vol. 2, No. 2, 1988, 151-169.

[CaGe88] N. Carriero and D. Gelernter, "Applications Experience with Linda," in *Proceedings of the ACM Symposium on Parallel Programming*, ACM, July 1988, 173-187.

[CaGe89] N. Carriero and D. Gelernter, "Linda in Context," in *CACM* 32, 4, April 1989, 444-458.

[Chow83] F. Chow, "A Portable Machine-Independent Global Optimizer," Technical Report, Number CSL-86-289, Standford University, May 1986.

[CoFe81] P. Cohen and E. Feigenbaum, "The Handbook of Artificial Intelligence," Vol. 3, William Kaufmann, 1981.

[CoKeTo88] K. Cooper, K. Kennedy, and L. Torczon, "Interprocedural Side-Effect Analysis in Linear Time," in *SIGPLAN Notices*, Vol. 21, No. 7, 1988, V57-66.

[Cytron84] R. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, Department of CS, University of Illinois, Urbana-Champaign Report No. UIUCDCS-R-84-1177, August 1984.

[Dally88] W. Dally, "Fine-Grain Message-Passing Concurrent Computers," in *Proceedings of the Third Hypercube Conference*, Vol. 1, ACM, 1988, 2-12.

[Dong87] Dongarra, J.J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," in W. J. Karplus (editor) *Multiprocessors and Array Processors*,

Simulation Councils Inc. San Diego, CA., January 1987, 15-33.

[EHLR80] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, "The HEARSAY II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Survey*, Vol. 12, No. 2, 1980, 213-253.

[Ellis85] J. Ellis, "Bulldog: A Compiler for VLIW Architectures," 1985 ACM Doctoral Dissertation Award, The MIT Press, 1986.

[EnTe79] R. Engelmore and A. Terry, "Structure and Function of the CRYSALIS System," in *Proceedings of Sixth International Joint Conference on Artificial Intelligence*, Vol. 1, Tokyo, Japan, August 1979, 250-256.

[ErLe75] L. D. Erman and V. R. Lesser, "A Multi-Level Organization for Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge," in *Proceedings of Fourth International Joint Conference on Artificial Intelligence*, Vol. 1, Tbilisi, USSR, 1975, 483-490.

[FeLe77] R. D. Fennell and V. R. Lesser, "Parallelism in Artificial Intelligence Problem Solving : A case Study of HEARSAY II," in *IEEE Transactions on Computers*, Vol. C26, No. 2, 1977, 98-111.

[Feng72] T. Y. Feng, "Some Characteristics of Associative/Parallel Processing," Proc. 1972 Sagamore Computer Conference, Syracuse University, 1972, 5-16.

[FeOtWa83] J. Ferrante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10543, August 1983.

[Fisher84] J. Fisher, "The VLIW machine: A multiprocessor for Compiling Scientific Code," *IEEE Computer*, July 1984, 45-53.

[Flynn66] M. J. Flynn, "Very High Speed Computing Systems," *Proceedings of IEEE, Vol. 54*, 1966, 1901-1909.

[FuNi88] K. Fuchi, and M. Nivat (editors), "Programming of Future Generation Computers," North-Holland, 1988.

[GGJMG89] V. Guarna Jr., D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur, "Faust: an Integrated Environment for the development of Parallel Programs," *IEEE software*, July 1989.

[GJMW89] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff, "Performance Prediction of Loop Constructs on Multiprocessor Hierarchical-Memory Systems," Technical Report, CSRD Rpt No. 853, Center for Supercomputer Research and Development, University of Illinois, 1989.

[GaVR84] D. Gannon and J. Van Rosendale, "On the Communication Complexity of Parallel Numerical Algorithms," *IEEE Transactions on Computers*, December 1984, C-33 #12, 1180-1194.

[GaJaGa87] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation," in *Proceedings of the 1987 International Conference on Supercomputing*, 1987, 229-254.

[Gerndt89] H.M. Gerndt, "Automatic Parallelization for Distributed-Memory Multiprocessing Systems," Ph.D. thesis, University Bonn, December 1989.

[Gerndt90] H. M. Gerndt, "Updating Distributed Variables in Local Computations," *Concurrency: Practice and Experience*, Vol 2(3), Sept. 1990, 171-193.

[GrWe76] S. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," in *JACM*, Vol. 23, No. 1, January 1976, 172-202.

[Grub89] T. Gruber, "The Acquisition of Strategic Knowledge," Academic Press, 1989.

[GFNW86] A. Gupta, C. Forgy, A. Newell, and R. Wedig, "Parallel Algorithm and Architectures for Rule-Based Systems," in *Proceedings of the 13th Symposium on Computer Architecture*, June 1986,

[HHRC79] B. Hayes-Roth, F. Hayes-Roth, S. Rosenschein, and S. Cammarata, "Modeling Planning as an Incremental, Opportunistic Process," in *Proceedings of Sixth International Joint Conference on Artificial Intelligence*, Vol. 1, Tokyo, Japan, August 1979, 375-383.

[Händler77] W. Händler, "The Impact of Classification Schemes on Computer Architecture," in *Proceedings of the 1977 International Conference on Parallel Processing*, 1977, 7-15.

[HaLe77] F. Hayes-Roth and V. R. Lesser, "Focus of Attention in the HEARSAY II Speech Understanding System," *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, Vol. 1, Boston, Massachusetts, USA, 1977, 27-35.

[Hayes79] B. Hayes-Roth and F. Hayes-Roth, "A Cognitive Model of Planning," *Cognitive Science*, Vol. 3, 1979, 275-310.

[Hayes83] B. Hayes-Roth, "The Blackboard Architecture: A General Framework for Problem Solving?," Technical Report, No. HPP-83-30, Department of Computer Science, Standford Univ., Standford, May 1983.

[Hayes85] B. Hayes-Roth, "A Blackboard Architecture for Control," in *Artificial Intelligence*, Vol. 26, No. 3, July 1985, 251-321.

[HePa90] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers, Inc., 1990.

[Hecht77] M. Hecht, "Flow Analysis of Computer Programs," North Holland, 1977.

[Hillis85] W. Hillis, "The Connection Machine," The MIT Press, 1985.

[HoRe77] R. Hon, and D.R. Reddy, "The Effects of Computer Architecture on Algorithm Decomposition and Performance," in Kuck, et al. (editors), *High-Speed Computers and Algorithm Organization* Academic Press, 1977, 411-421.

[Husm86] H. Husmann, "Compiler Memory Management and Compound Function Definition for Multiprocessors," Ph.D. Thesis, Department of Computer Science, University of Illinois, CSRD Rpt. No. 575.

[Hwang84] K. Hwang "Supercomputers - Design and Applications," McGraw-Hill, 1984.

[HwBr84] K. Hwang and F. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, 1984.

[HwDe89] K. Hwang and D. DeGroot (editors), "Parallel Processing For Supercomputers & Artificial Intelligence," McGraw-Hill, 1989.

[Intel90] Intel Corporation, "iPSC/2 and iPSC/860 User's Guide," Intel Corporation, June, 1990.

[KGSF84] A. Kapauan, D. Gannon, L. Snyder, and T. Field, "The Pringle Parallel Computer," in *Proceedings of the 11th International Symposium on Computer Architecture*, IEEE, 1984, 12-20.

[KWGCS84] A. Kapauan, K. Wang, D. Gannon, J. Cuny, and L. Snyder, "The Pringle: An Experimental System for Parallel Algorithm Design and Testing," in *Proceedings of the 1984 International Conference on Parallel Processing*, 1984, 1-8.

[KaUl76] J. Kam and J. Ullman, "Global Data Flow Analysis and Iterative Algorithms," in *JACM*, Vol. 23, No. 1, January 1976, 158-171.

[Karp87] A. Karp, "Programming For Parallelism," in *Computer*, May 1987, 43-57.

[Kenn80] K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report 476-029-4, Rice University, October 1980

[Kers88] L. Kerschberg (editor), "Expert Database Systems," Benjamin/Cunnubgs, 1988.

[KlWa86] P. Klahr and D. Waterman (editors), "Expert Systems: Techniques, Tools, and Applications," Addison Wesley, 1986.

[Koel90] C. Koelbel, "Compiling Programs For Non-Shared Memory Machines," Ph.D. thesis, Department of Computer Science, Purdue University, December 1990. Technical report no. CSD-TR-1037, 1990.

[KoMe89] C. Koelbel and P. Mehrotra, "Compiler Transformations for Non-Shared Memory Machines," in *Proceedings of the 4th International Conference on Supercomputing*, Vol. 1, 1989, 390-397.

[KoMeVR90] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Supporting Shared Data Structures on the Distributed Memory Architectures," in *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, 1990, 177-186.

[Kowa85] J. Kowalik, "Parallel MIMD Computation: Hep Supercomputer and Its Applications," The MIT Press, 1985.

[KKLW80] D. Kuck, R. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," in *Proceedings of the 4th International Computer Software and Application Conference*, October 1980, 709-715.

[KKLPW81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. H. Padua and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th Annual ACM Symposium on Principles Of Programming Languages*, Williamsburg, VA., January 1981.

[KuWM84] D. Kuck, M. Wolfe, and J. McGraw, "A Debate: Retire FORTRAN?," in *Physics Today*, May 1984, 67-75.

[KDLS86] D. Kuck, E. Davidson, D, Lawrie, and A. Sameh, "Parallel Supercomputing Today and the Cedar Approach," in *Science* Vol. 231, February 1986, 967-974.

[LeCo83] V.R. Lesser and D.D. Corkill, "The Distributed Vehicle Monitoring Test Bed," in *AI Magazine*, Fall 1983, 15-33.

[LeEr77] V. R. Lesser and L. D. Erman, "A Retrospective View of the HEARSAY II Architecture," in *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, Boston, USA, 1977, 790-800.

[Leung90] B. Leung, "Issues on the Design of Parallelizing Compilers," CSRD Report No 1012, Center for Supercomputer Research and Development, University of Illinois, June 1990.

[LeWi89] J. Levesque, and J. Williamson, "A Guidebook to Fortran on Supercomputers," Academic Press, 1987.

[LiYe88a] Z. Li and P. Yew, "Interprocedural Analysis for Parallel Computing," in *Proceedings of the International Conference on Parallel Processing*, 1988.

[LiYe88b] Z. Li and P. Yew, "Efficient Interprocedural Analysis for Parallel Parallelization and Restructuring," Technical Report, Center for Supercomputer Research and Development, CSRD Rpt. No. 804.

[MeVR87] P. Mehrotra, J. R. Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming," in *Parallel Computing*, Vol. 5, 1987, 339-361.

[MeVR89a] P. Mehrotra and J. Van Rosendale, "Compiling High Level Constructs to Distributed Memory Architectures," in *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[MeVR89b] P. Mehrotra and J. Van Rosendale, "Parallel Language Constructs for Tensor Product Computations on Loosely Coupled Architectures," in *Proceedings of Supercomputing '89*, Reno NV, Nov. 1989, 616-626.

[Myer81] E. Myers, "A Precise Inter-Procedural Data Flow Algorithm," in *Proceedings of the 8th Annual ACM Symposium on Principle of Programming Languages*, 1981, 219-230.

[MiYa87] B. Miller and C. Yang, "IPS: an Interactive and Automatic Performance Tool for Parallel and Distributed Programs," in *Proceedings of the 7th International Conference on Distributed Computing Systems*, 1987, 482-489.

[Minsky75] M. Minsky, "A Framework for Representing Knowledge," in P. Winston (editor), *The Psychology of Computer Vision*. McGraw-Hill, 1975, 211- 277.

[MiPa90] S. Midkiff and D. Padua, "Issues in the Compile-Time Optimization of Parallel Programs," Technical Report, Center for Supercomputer Research and Development, Rpt. No. 993, University of Illinois, May 1990.

[Nau83] D. Nau, "Expert Computer Systems," in *IEEE Computer*, February, 1983, 63-85.

[NCUBE87] NCUBE Corporation, "NCUBE Users Handbook," NCUBE Corporation, 1987.

[NCUBE90] NCUBE Corporation, "nCUBE 2 Processor Manual," NCUBE Corporation, 1990.

[Nii85] H. P. Nii, "Research on Blackboard Architecture at the Heuristic Programming Project," Technical Report No. KSL-85-24, Department of Computer Science, Standford University, Standford, May 1985.

[Nii86a] H. P. Nii, "CAGE and POLIGON: Two frameworks for Blackboard based Concurrent Problem Solving," Technical Report No. KSL-86-41, Department of Computer Science, Standford University, Standford, April 1986.

[Nii86b] H. P. Nii, "Blackboard Systems: The Blackboard Model of Problem Solving and Evolution of Blackboard Architectures," Part 1, in *AI Magazine*, August 1986, 38-53.

[Nii86c] H. P. Nii, "Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective," Part 2, in *AI Magazine*, August 1986, 82-106.

[Nilsson80] N. J. Nilsson, "Problem-solving Methods in Artificial Intelligence," McGraw-Hill, 1980.

[Padua79] D. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois, Urbana-Champaign, November 1979.

[PaGuLa87] D. Padua, V. A. Guarna Jr. and D. Lawrie "Supercomputer Programming Environments," in *Parallel Computations and Their Impact on Mechanics*, Vol. 86, December 1987, 55-79.

[PaKu80] D. Padua and D. Kuck, "High-Speed Multiprocessors and Compilation Techniques," in *IEEE Transactions on Computers*, Vol. C-29, No. 9, September, 1980, 763-776.

[Para89] Parasoft Co., "EXPRESS: A Communication Environment for Parallel Computers," Parasoft Corporation, 27415 Trabuco Circle, Mission Viejo, CA., 1988.

[PfBrNo85] G. Pfister, W. Brantley, D. George, S. Harvey, and W. Kleinfelder, K. McAuliffe, E. Melton, V.A. Norton, J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,", in *Proc. of the 1985 International Conference on Parallel Processing*, 1985, 764-771.

[PfNo85] G. Pfister, V.A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," in *Proc. of the 1985 International Conference on Parallel Processing*, 1985, 790-797.

[Poly86] C. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. Thesis, University of Illinois Center for Supercomputer Research and Development, CSRD TR-595, August 1986.

[Poly88] C. Polychronopoulos, "Toward Auto-Scheduling Compilers," CSRD Report No. 789, Center for Supercomputer Research and Development, University of Illinois, May 1988.

[PGHLLS89] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," in *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[RoPi90] A. Rogers and K. Pingali, "Process Decomposition through Locality of Reference," in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[RoScWe89] M. Rosing, R. Schnabel, and R. Weaver, "Expressing Complex Parallel Algorithms in DINO," in *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers and Applications*, 1989, 553-560.

[RüAn88] R. Rühl and M. Annaratone, 'Parallelization of Fortran Code on Distributed-Memory Parallel Processors," in *Proceedings of the 1990 International Conference on Supercomputing*, June, 1990, 342-353.

[Sacer77] E.D. Sacerdoti, "A Structure for Plans and Behavior," Elsevier Science Publishers, 1977.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-Time Scheduling and Execution of Loops on Message Passing Machines," in *Journal of Parallel and Distributed Computing*, Vol. 8, No. 4, April 1990, 303-312.

[Sarkar87] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors, Ph.D. thesis, Stanford University, 1987.

[Schw80] J. Schwartz, "Ultracomputer," in *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, October 1980, 484-521.

[ScRu85] Z. Segall and L. Rudolph, "PIE: a Programming and Instrumentation Environment for Parallel Processing," in *IEEE Software*, Vol. 2, No. 6, November 1985.

[Smith82] A. Smith, "Cache Memories," in *Computing Surveys*, Vol. 14, No. 3, September 1982, 473-530.

[Soren] D. Sorensen, Personal communication.

[Stone90] H. Stone, "High-Performance Computer Architecture, Second Edition" Addison Wesley, 1990.

[Tate76] A. Tate, "Project Planning Using a Hierarchic Non-Linear Planner," Technical Report No. 25, Department of Artificial Intelligence, University of Edinburgh, 1976.

[Terr83] A. Terry, "The CRYSALIS Project: Hierarchical Control of Production Systems," Memo HPP-83-19, Department of Computer Science, Standford University, Standford, May 1983.

[TsLaKu88] P.S. Tseng, M. Lam, and H.T. Kung, "The Domain Parallel Computation Model on Warp," in *Proceeding of SPIE*, SPIE, 1988.

[Veid85] A. Veidenbaum, "Compiler Optimizations and Architecture Design Issues For Multiprocessors," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, CSRD Rpt. No. 520, Center for Supercomputer Research and Development, 1985.

[WMSW87] A. Walker, M. McCord, J. Sowa, and W. Wilson, "Knowledge Systems and Prolog," Addison-Wesley, 1987.

[Wang85] K. Wang, "An Experiment in Parallel Programming Environment: The Expert Systems Approach," in K. S. Fu (editor), *Some Prototype Examples for Expert Systems*, TR-EE 85-1, Purdue University, March 1985, 591-624.

[Wang85b] K. Wang, "A Fast Program Dependence Analysis Algorithm for Blaze," CS690V Class Report, Department of Computer Science, Purdue University, August 1985.

[Wang86] K. Wang, "Array Protection and Copying Issues in the Blaze and E-Blaze Implementation," Internal memo, the RP3 group, IBM, August 1986.

[Wang88] K. Wang, "Machine Knowledge Representation and Manipulation For Parallel Compilers," Technical Report CSD-TR-843, Department of Computer Sciences, Purdue University, December 1988.

[WaGa89] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimizations For Parallel Computers," in K. Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence*, McGraw-Hill, 1989, 441-485.

[Wang90a] K. Wang, "A Performance Prediction Model For Parallel Compilers," Technical Report, CSD-TR-1041, CAPO Report, CER-90-43, Department of Computer Science, Purdue University, November 1990.

[Wang90b] K. Wang, "Array Reshaping - A Mechanism for Optimizing Array Storage On Parallel Architectures," Technical Report, CSD-TR-1042, CAPO Report, CER-90-44, Department of Computer Science, Purdue University, November 1990.

[Wang90c] K. Wang, "Managing Data Synchronization Automatically For Distributed-Memory Architectures," Technical Report, CSD-TR-1043, CAPO Report CER-90-45, Department of Computer Science, Purdue University, November 1990.

[Wang90d] K. Wang, "A Framework For Intelligent Parallel Compilers," Technical Report, CSD-TR-1044, Department of Computer Science, Purdue University, November 1990.

[Wang90e] K. Wang, "Heuristic Guided Pre-Optimized Algorithm Substitution For Parallel Computers," Technical Report, CSD-TR-1055, CAPO Report, CER-90-50, Department of Computer Science, Purdue University, December 1990.

[WeKu84] S. Weiss and C. Kulilowski, "A Practical Guide to Designing Expert Systems," Rowman and Allanheld publishers, 1984.

[Wilk84] D. Wilkins, "Domain Independent Planning: Representation and Plan Generation," in *Artificial Intelligence*, Vol. 22, 1984, 269-301.

[Will85] M.A. Williams, "Distributed, Cooperating Expert Systems for Signal Understanding," In *Proceedings of Seminar on AI Application to Battlefield*, 341-346.

[Wino75] T. Winograd, "Frame Representations and the Declarative/Procedural Controversy," in Bobrow and Collins (editors), *Representation and Understanding: Studies in Cognitive Science*, Academic Press, 1975, 185-210.

[Wolfe82] M. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1982, Report no. UIUCDCS-R-82-1105.

[Wolfe89] M. Wolfe, "Optimizing Supercompilers for Supercomputers," The MIT Press, 1989.

[Yew88] P-C Yew, "Architecture of the Cedar Parallel Supercomputer," in *Parallel Systems and Computation* Paul and G.S. Almasi (editors), Elsevier Science Publishers, 1988, 137-148.

[ZiBaGe88] H. P. Zima, H.-J. Bast and H. M. Gerndt, "SUPERB: A Tool for Semi-automatic MIMD/SIMD Parallelization," *Parallel Computing*, Vol 6, 1988, 1-18.

[ZiCh90] H. Zima and B. Chapman, "Supercompilers for Parallel and Vector Computers," Addison-Wesley publishing, 1990.

APPENDIX

Appendix A.1. Sample Rules Used In Chapter 8

Process Creation.

```
[Rule a.1, ['computational model construction']]
if (has 'self-scheduling-loop primitives')
then
   assert('parallelize outermost loop without blocking').

[Rule a.2, ['computational model construction']]
if ('process creation cost' is high) and
   (number-of-processors is P)
then
   assert('number of processes to create' is P).

[Rule a.3, ['computational model construction']]
if ('process creation cost' is low)
then
   assert('parallelize outermost loop without blocking').
```

Locality

```
[Rule a.4, ['computational model construction']]
if has-cache
then
   assert('locality is important').

[Rule a.5, ['computational model construction']]
if ('data access/process cost ratio' is high)
then
   assert('memory optimization dominates instruction minimization').

[Rule a.6, ['computational model construction']]
if ('shared/local memory access ratio' is high)
then
   (assert('locality is important')) and
   (assert('use local variable whenever possible')).

[Rule a.7 ['computational model construction']]
if (has 'vector register')
then
   (try 'keep vector operand in register')
```

The Program Focus Selection Subgoal

```
[Rule b.1, ['program focus selection']]
if ('nested loop'(PDG)) and
   (nested-in(BB, PDG)) and
   ('single statement block'(BB))
then
   FOCUS = PDG.
```

The Transformation Selection Subgoal.

```
[Rule c.1, ['program restructuring subgoal selection']]
```

```
if ('nested loop'(FOCUS)) and
    (nested-in(BB, FOCUS)) and
    ('single statement block'(BB))
then
    select('task creation and processor allocation').

[Rule c.2, ['program restructuring subgoal selection']]
if ('compound statement'(Focus))
then
    select('parallelism improvement').

[Rule c.3, ['program restructuring subgoal selection']]
if ('tasks created')
then
   select('memory access optimization').

[Rule c.4, ['program restructuring subgoal selection']]
:- (select('task creation and processor allocation')).

[Rule c.5, ['program restructuring subgoal selection']]
if (('has cache') or ('has arrays in'(Focus)) or ('locality is important'))
then
    select('memory access optimization').

[Rule c.6, ['program restructuring subgoal selection']]
if ('multiple tasks are created')
then
    select('parallelism improvement').

[Rule c.7, ['program restructuring subgoal selection']]
if (('task created'(FOCUS)) and (not 'parallelized'(FOCUS)))
then
    select('task-creation and processor allocation')
```

Parallelism Improvement Subgoal

```
[Rule d.1, ['parallelism improving']]
if (is-a-loop(L)) and
    (L = (for i in [RANGE] do A += B[i] * C[i]; end for))
then
    is-inner-product(L)

[Rule d.2, ['parallelism improving']]
if ('has built-in fast inner product') and
    (is-in(L, FOCUS)) and
    (is-inner-product(L))
then
    apply(transformation(inner-product, L)).

[Rule d.3, ['parallelism improving']]
if ('has fetch-and-op operations') and
    ('recurrence relation'(STMT))
then
    ('change into accumulation'(STMT)).

[Rule d.4, ['parallelism improving']]
If ('nested-loops'(Focus)) and
    (not 'perfectly-nested-loops'(Focus)) and
    (('is-multi-processors' and high('task-creation-time')) or
      ('has vector operations'))
then
    apply('loop distribution').
```

Rules about task creation and processor allocation

```
[Rule e.1, ['task creation and processor allocation']]
if (is-nested-loop(FOCUS))
```

```
then
   select(loop-interchange(FOCUS)).

[Rule e.2, ['task creation and processor allocation']]
if (is-nested-loop(FOCUS)) and
   ('has vector operations') and
   ('size of vector registers'(V)) and
   (V <> 0) and
   (innermost-loop(FOCUS, INNER)) and
   (num-of-iterations(INNER, N)) and
   (V < N)
then
   'loop blocking'(INNER, N).

[Rule e.3, ['task creation and processor allocation']]
if (is-a-loop(FOCUS)) and
   (outermost-loop(FOCUS, OUTER)) and
   (num-of-iterations(OUTER, N)) and
   (number-of-processor(P)) and
   (N > P)
then
   'loop blocking'(OUTER, P).

[Rule e.4, ['task creation and processor allocation']]
if ('parallelize outermost loop without blocking') and
   (is-nested-loop(FOCUS)) and
   (outermost-loop(FOCUS, OUTER))
then
   (parallelize(OUTER)).
```

Memory Access Optimization.

```
[Rule f.1, ['memory access optimization']]
(Assume L2 is the innermost loop that is nested in L1 such
that array references of X depends on the loop index of L2.
Also let X-sub be the part of the array X whose references
depend on the loops inside L2.)

if (has-instruction(block-transfer)) and
   (shared-array(X)) and
   (parallelize(L1)) and
   (referenced-in(X, L1)) and
   (innermost-depends-on-loop(L1, X, L2)) and
   (sub-depends-on(X, X-sub, L2)) and
   (N = sizeof(X)) and
   ('minimal number of references to justify cost of block-transfer' = B) and
   (N > B)
then
   (apply('block transfer'(X-sub, L2))).

[Rule f.2, ['memory access optimization']]
if (apply('block transfer'(X, L)) and
   (parallelize(L)) and
   ('nested in'(L1, L)) and
   (sub-depends-on(X, X-sub, L1))
then
   (apply('block transfer'(X-sub, L1))).

[Rule f.3, ['memory access optimization']]
if (apply('block transfer'(X, L))) and
   ('nested in'(L, LO))
then
   ('create temporary array'(tmp, LO) and
   ('create statement'(S, block-transfer(X, tmp, sizeof(X))) and
   ('insert in front of'(S, L2)) and
   (substitute(X, tmp, L)).
```

```
[Rule f.4, ['memory access optimization']]
if (S = ('block transfer'(A, L, N))) and
    (shared(A)) and
    (local(L)) and
    (nested-in(S, L0)) and
    (parallelized(L0)) and
    ('not depends on'(A, L0)) and
    ('number of processors'(P))
then
    (create-loop(LL, 1..P)) and
    (add-stmt(LL, S)) and
    (parallelized(LL)) and
    ('insert in front of'(LL, L0)).

[Rule f.5, ['memory access optimization']]
if (S = ('block transfer'(L, A, N))) and
    (shared(A)) and
    (local(L)) and
    (nested-in(S, L0)) and
    (parallelized(L0)) and
    ('not depends on'(A, L0)) and
    ('number of processors'(P))
then
    (create-loop(LL, 1..P)) and
    (add-stmt(LL, S)) and
    (parallelized(LL)) and
    ('append to'(LL, L0)).

[Rule f.6, ['memory access optimization']]
if ('has cache') and
    ('mostly used array'(A, FOCUS))
then
    ('keep in cache'(A)).

[Rule f.7, ['memory access optimization']]
if ('locality is important') and
    ('has local memory') and
    ('data accessing ratio of shared memory-local memory' > 2) and
    (shared-array(A))
then
    ('allocate array A to the local memory of each processor').

[Rule f.8, ['memory access optimization']]
if (has-local-memory)
    ('mostly used array'(A, FOCUS))
    (shared-array(A))
    (appears-in(A, S)) and
    ('in nested loops'(S, [L1 .. Ln])) and
    ('not depends on loops'(A, L1))
then
    ('create tmp'(tmp, L1)) and
    ('create statement'(S1, (A := tmp))) and
    ('insert in front of'(S1, S)),
    (substitute(A, tmp, L1)).

[Rule f.9, ['memory access optimization']]
if ('mostly used array'(A, FOCUS)) and
    (shared(A)) and
    (appears-in(A, S)) and
    ('in nested loops'(S, [L1.. Ln])) and
    ('depends on loops'(A, L1))
then
    (find the plausible loop order ORD with most inner loops that A depends on) and
    ('loop interchange'(L1, ORD)) and
    (innermost-depends-on-loop(L1, X, LL)) and
    ('create tmp'(tmp, LL)) and
    ('create statement'(S1, (A := tmp))) and
```

```
    ('insert in front of'(S1, S)) and
    (substitute(A, tmp, LL)).

[Rule f.10, ['memory access optimization']]
if ('has local memory') and
    ('mostly used array'(A, FOCUS)) and
    (shared(A)) and
    ('not modified'(A)) and
    (cache-size(C)) and
    (sizeof(A) > C)
then
    ('create tmp'(tmp, FOCUS)) and
    (scalarize(A, tmp)).

[Rule f.11, ['memory access optimization']]
if ('non-shared memory') and
    (parallelized(L)) and
    (array(A)) and
    (appears-in(A, L)) and
    ('no inter task dependence exist'(A, L)) and
    (sub-depends-on(A, A-sub, L))
then
    (allocate-local(A-sub, L)).

[Rule f.12, ['memory access optimization']]
if ('non-shared memory') and
    (parallelized(L)) and
    (array(A)) and
    (appears-in(A, L)) and
    ('has inter task dependence in'(A, L))
then
    ('pipelining references'(A, L)).

[Rule f.13, ['memory access optimization']]
if ('has vector register') and
    ('is a vector'(V)) and
    (appears-in(V, S)) and
    ('in nested loops'(S, LList)) and
    (member(LL, LList)) and
    ('not depends on'(A, LL))
then
    ('interchange loops to move LL into the innermost').

[Rule f.14, ['memory access optimization']]
if ('has vector register')
then
    ('vector register optimization dominates memory access optimization')
```

Appendix A.2. Sample Listing of Encoded Rules

```
% File: tran/cshrinking.a.
%       rules for cycle shrinking.
% Cycle shrinking is a special case of loop blocking!
% We can use cycle shrinking to squeeze parallelism out of serial loop.
%       For heuristics and rules see file cshrinking.
% We actually use loop_blocking(Loop, parallel_row2, ReductionFactor)
%       to perform cycle shrinking.
%       For non-perfectly nested loops, loop unrolling for the first or last
%       iteration of the inner loop may be needed.
%       For true-dependence shrinking, we may be better of by collapsing the
%       loops into one single loop and block the resulting loop.

% module(cycle_shrinking).
% transformation_file(lblocking).
%       loop blocking is used to squeeze parallelism out of serial loops.
```

```
% Loop is the outermost loop of a nested loop (perfect or non-perfect)

% Rule 1.
% if   apply_transformation(cycle_shrinking, Loop, Type, ReductionFactor)
% then call(perform_transformation(cycle_shrinking, [Loop, Type, ReductionFactor]))
rule(1, cycle_shrinking,
       apply_transformation(cycle_shrinking, Loop, Type, ReductionFactor),
       call(perform_transformation(loop_blocking, [Loop, Type, ReductionFactor])),
       1.0).

% Rule 2.
% if    serial(Loop) and          /* form a strongly connected graph */
%       all_distance_are_known(Loop)
% then
%       try_transformation(loop_shrinking, Loop, BlockingFactor)
rule(2,cycle_shrinking,
      (serial(Loop), no_unknown_distance(Loop)),
      try_transformation(cycle_shrinking, Loop, Type, ReductionFactor),
      1.0).

% Rule 3.
%     if try_transformation(loop_shrinking, Loop, D) and
%       not_nested_loop(Loop) and     /* there is no loops nested inside  */
%       D = min{DDep: distance(Dep, DDep),
%             for all cross loop dependence Dep in Loop}
%       then
%       apply_transformation(loop_shrinking, Loop, D).
%/* ie   apply('loop blocking', parallel_row2) and parallelize(new_inner) */
rule(3, cycle_shrinking,
       (try_transformation(cycle_shrinking, Loop, simple, ReductionFactor),
        call(single_loop(Loop)),
        call(simple_cycle_shrinking_arg(Loop, ReductionFactor))
       ),
       apply_transformation(loop_shrinking, Loop, simple, ReductionFactor),
       1.0).

% Rule 4.
%       Assuming DistanceVector = (d1, d2, ..., dn) has the minimum true-distance
%       in loop Loop, d sub r1 is the first postive distance in D, and
%     loop Loop is a perfect nested loop,
%   if there is a r2 s.t.
%       all dr = 0 for r in (r1+1, ..., r2-1) and dr2 < 0
%   then apply selective shrinking is better than apply true distance shrinking
rule(4, cycle_shrinking,
       (try_transformation(cycle_shrinking, Loop, selective, ReductionFactor),
        call(perfect_nested_loops(Loop, LoopList)),
        call(min_true_distance(LoopList, DistanceVector, TrueDistance)),
        call(selective_shrinking_is_better(DistanceVector, ReductionFactor))
       ),
       apply_transformation(loop_shrinking, Loop, selctive, ReductionFacrtor),
       1.0).

% Rule 5.
%       For the same assumption as in rule 4.
%   if there is no r2 s.t.
%       all dr = 0 for r in (r1+1, ..., r2-1) and dr2 < 0
%   then apply true distance schinking is better,
rule(5, cycle_shrinking,
       (try_transformation(cycle_shrinking, Loop, true_dependence, ReductionFactor),
        call(perfect_nested_loops(Loop, LoopList)),
        call(min_true_distance(LoopList, DistanceVector, ReductionFactor)),
        call(+ selective_shrinking_is_better(DistanceVector, _))
       ),
       apply_transformation(loop_shrinking, Loop, true_dependence, ReductionFactor),
       1.0).

% Rule 6.
```

```
% if the loop is not perfectly nested:
%       only the common outer loop is considered.
%       the loops inside the outer loop can be considered seperately.
rule(6, cycle_shrinking,
        (try_transformation(cycle_shrinking, Loop, non_perfect_nested, ReductionFactor),
         call(+ simple_loop(Loop)),
         call(+ perfect_nested_loops(Loop, _)),
         call(min_distance(Loop, ReductionFactor))
        ),
        apply_transformation(loop_shrinking, Loop, non_perfect_nested, ReductionFactor),
        1.0).
```

Appendix B. Program Transformation Techniques

* Transformations included:
    See table 8.1.

* Steps to Add New Transformations To the System.

1. Implement the transformation in three modules:
   b. applicability test - routine to test if the transformation is applicable.
      'transformation_name'_test(Focus, Arg).
   b. argument selection - routine to find suitable argument for the transformation:
      'transformation_name'_arg(Focus, Arg).
   c. algorithm application - routine to perform the transformation:
      'transformation_name'(Focus, Arg).

2. Declare the transformation.
   In file tran/transformation add the predicate:
      transformation(transformation_name, SubGoal, Availbility).
   where Availbility is the status of the transformation, its value is either
      ok, not_reliable, partial, or no.
   Example:
      transformation(loop_interchanging, task_creation, ok).

3. Add rules for transformation selection:
   put the rules to select transformations in select_transformation/6 in
   file tran/transformation.

4. Add rules for making things happen
   add rules to make the transformation appllicable when it is not applicable to the program
   these rules should be added to the file tran/transformation.
      try_harder_'transformation_name'(Focus, SubGoal, Arg).

5. Add specialized evaluation functions and weight to the file
      evaluation_function_for('transformation_name', Goal, EvalFunctions).
      weight_evaluation('transformation_name', Goal, EvalFunction, Weight).
   Example:
      evaluation_function_for(loop_parallelization, _,
         [loop_size_matching,doacross_delay,array_stride,locality,synchronization_factor]).
      weight_evaluation(loop_interchange, _, array_stride, Weight) :-
         has_vector_capability,
         (feature_value(operand_stores, in_memory) ->
            Weight = 1.0
         ;
            Weight = 0.5
).
```
```